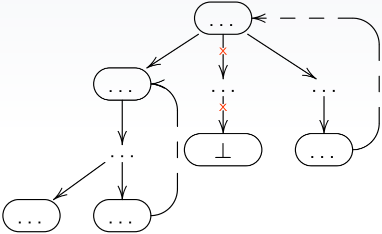
Experiments with Supercompilation on Refal

Antonina Nepeivoda
Program Systems Institute of RAS

ru-$\mathcal{STEP}$, Innopolis, July 9th

Introduction to the Supercompilation



Considers the set of all runs of the program on a given parameterized entry point.

- unfolding: general case $\longrightarrow$ a set of specific cases;
- and folding: specific case $\longrightarrow$ another specific case or more general one.

We denote $\mathtt{e}$-parameters with u, w; and $\mathtt{s}$-parameters with s.

Introductory Example

```
GenA {
  /* EMPTY */  =  /* EMPTY */;
  s.x e.y      =  A <GenA e.y>;
}
AllA {
  /* EMPTY */  =  T;
  A e.x        =  <AllA e.x>;
  s.x e.y      =  F;
}
```

The parameterized entry point: $<\mathtt{AllA}\ <\mathtt{GenA}\ w>>$.

Supercompiling <AllA <GenA *w*>>

$$\begin{array}{c} \boxed{<\underline{\texttt{AllA}}<\texttt{GenA}\ w>>} \\ \downarrow \\ \boxed{<\texttt{AllA}<\underline{\texttt{GenA}}\ w>>} \end{array}$$

Refal uses CBV semantics; supercompilation uses also CBN.

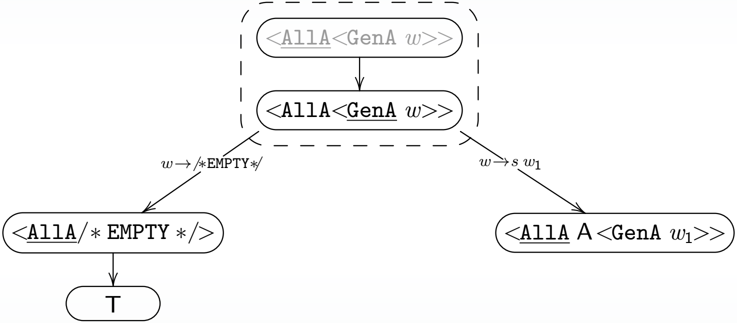
Driving: One-Step Unfolding

Definition

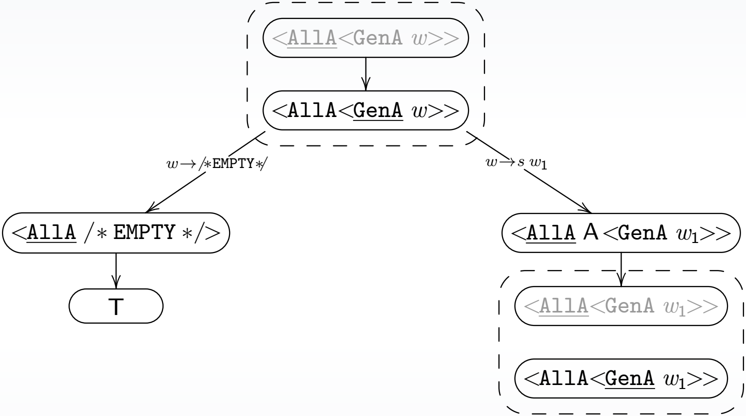
One-step unfolding of the state C_0 is a transition to the states $C_1, \dots, C_n$ such that every computation path starting in C_0 is a path starting in C_i, prefixed with the state C_0.

Given the (parameterized) call $f(t_1, \dots, t_k)$ and the f definition consisting of n rules $f(P^i_1, \dots, P^i_k) = R_i$, driving generates substitution pairs $\langle \sigma_j, \xi_j \rangle$ (if possible) unifying $f(t_1, \dots, t_k)$ with $f(P^i_1, \dots, P^i_k)$, thus the general case $f(t_1, \dots, t_k)$ is specified to j cases $R_i \xi_j$, and the transitions are marked by the parameter narrowings σ_j.

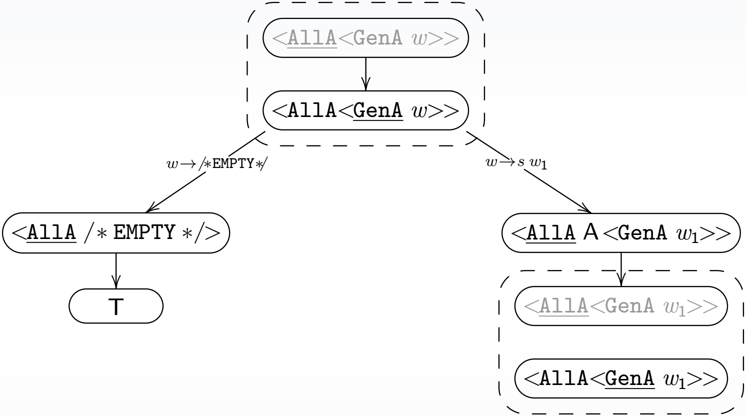
Supercompiling <AllA <GenA w>>



Supercompiling <AllA <GenA w>>

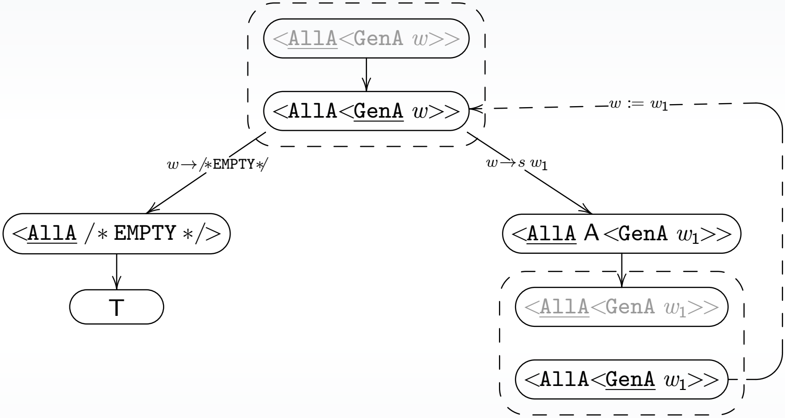


Supercompiling $<\texttt{AllA}<\texttt{GenA}\ w>>$



State $<\texttt{AllA}<\texttt{GenA}\ w_1>>$ repeats $<\texttt{AllA}<\texttt{GenA}\ w>>$ modulo par-renaming.

Supercompiling <AllA <GenA w>>



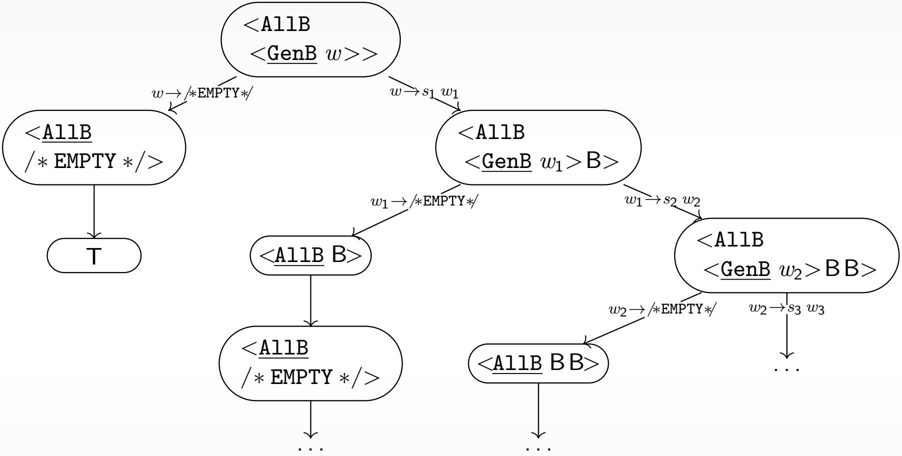
All the cases considered are either object expressions or folded to the known ones.

Another example: <AllB <GenB *w*>>

```
GenB {
   /* EMPTY */   =   /* EMPTY */;
   s.x e.y       =   <GenB e.y> B;
}
AllB {
   /* EMPTY */   =   T;
   B e.x         =   <AllB e.x>;
   s.x e.y       =   F;
}
```

The function `GenB` uses another order of the concatenation.

Another example: <AllB <GenB w>>



No par-substitution can be built. The path requires generalization.

Folding and Generalization

Definition

State C_1 ***is embedded*** *in state* C_2 *of the process graph, iff all the computation paths generated by* C_1 *are generated by* C_2.

Definition

State C_g ***generalises*** *states* C_1 *and* C_2 *of the process graph, iff all the computation paths generated by both* C_1 *and* C_2 *are generated also by* C_g.

Folding and Generalization

Definition

State C_1 ***is embedded*** *in state* C_2 *of the process graph, iff* $\exists\sigma$ *s.t.* $C_2\sigma = C_1$.

Definition

State C_g ***generalises*** *the states* C_1 *and* C_2 *of the process graph, iff* $\exists\sigma_1, \sigma_2$ *s.t.* $C_g\sigma_1 = C_1$, $C_g\sigma_2 = C_2$.

- Syntactic
- Easy-to-check

Supercompiling <AllB <GenB w>>

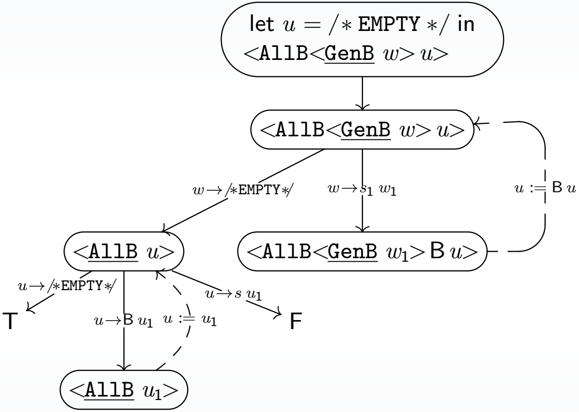
Definition

State C_g ***generalises*** *the states* C_1 *and* C_2 *of the process graph, iff* $\exists \sigma_1, \sigma_2$ *s.t.* $C_g\sigma_1 = C_1$, $C_g\sigma_2 = C_2$.

We require C_1 and C_2 to belong to the same path in the partial process graph.

After the generalization, the path unfolded from C_1 is deleted, and C_1 is replaced by the let-node: let σ_1 in C_g. The state C_g and the states corresponding to the rhs of σ_1 are unfolded independently.

Supercompiling <AllB <GenB w>>



Driving: One-Step Unfolding

Aim: solving equations $\mathtt{E} : \mathtt{P_i}$, where $\mathtt{E}$ is a parameterized expression; $\mathtt{P_i}$ — pattern.

Lisp-patterns — trees with non-repeated variables.

Lisp-data SCP	*Refal SCP*

Driving: One-Step Unfolding

Aim: solving equations $\mathtt{E} : \mathtt{P}_\mathtt{i}$, where $\mathtt{E}$ is a parameterized expression; $\mathtt{P}_\mathtt{i}$ — pattern.

Lisp-patterns — trees with non-repeated variables.

Lisp-data SCP	*Refal SCP*
linear unification problem solved by well-known unification algorithms $\text{Answer} \begin{cases} \text{contradiction} \\ \text{substitution } \sigma \\ \text{DEMAND} \end{cases}$	

Driving: One-Step Unfolding

Aim: solving equations $\mathtt{E : P_i}$, where $\mathtt{E}$ is a parameterized expression; $\mathtt{P_i}$ — pattern.

Lisp-patterns — trees with non-repeated variables.

Lisp-data SCP	*Refal SCP*
linear unification problem solved by well-known unification algorithms Answer $\begin{cases} \text{contradiction} \\ \text{substitution } \sigma \\ \text{DEMAND} \end{cases}$	*nested-word equation problem* no well-known algorithms Answer $\begin{cases} \text{contradiction} \\ \text{list } \langle \sigma_i \rangle\text{???} \\ \text{DEMAND} \end{cases}$

Complex Driving Example

```
Eq {
  (e.x)(e.x)  =  T;
  (e.x)(e.y)  =  F;
}
```

Driving <Eq (A w)(w A)>

$\rightarrow$ solving "unification problem" $\{$e.x : A w, e.x : w A$\}$

$\rightarrow$ representing solutions of A $w = w$ A

$\rightarrow$ $w \in \mathsf{A}^*$.

Problem: no finite set of substitutions can specify A^*.

Folding: Embedding and Generalization

Definition

State C_1 ***is embedded*** *in state* C_2 *of the process graph, iff* $\exists\sigma$ *s.t.* $C_2\sigma = C_1$ *(*$C_2 \preceq C_1$*).*

Definition

State C_g ***generalises*** *the states* C_1 *and* C_2 *of the process graph, iff* $\exists\sigma_1, \sigma_2$ *s.t.* $C_g\sigma_1 = C_1$, $C_g\sigma_2 = C_2$ *(i.e.* $C_g \preceq C_1$ & $C_g \preceq C_2$*).*

Lisp-data SCP	*Refal SCP*

Folding: Embedding and Generalization

Definition

State C_1 ***is embedded*** *in state* C_2 *of the process graph, iff* $\exists\sigma$ *s.t.* $C_2\sigma = C_1$ *(*$C_2 \preceq C_1$*).*

Definition

State C_g ***generalises*** *the states* C_1 *and* C_2 *of the process graph, iff* $\exists\sigma_1, \sigma_2$ *s.t.* $C_g\sigma_1 = C_1$*,* $C_g\sigma_2 = C_2$ *(i.e.* $C_g \preceq C_1$ & $C_g \preceq C_2$*).*

Lisp-data SCP	*Refal SCP*
anti-unification problem aka most specific generalization solved by well-known algorithm since (Plotkin, 1970)	

Folding: Embedding and Generalization

Definition

State C_1 ***is embedded*** *in state* C_2 *of the process graph, iff* $\exists\sigma$ *s.t.* $C_2\sigma = C_1$ *(*$C_2 \preceq C_1$*).*

Definition

State C_g ***generalises*** *the states* C_1 *and* C_2 *of the process graph, iff* $\exists\sigma_1, \sigma_2$ *s.t.* $C_g\sigma_1 = C_1$, $C_g\sigma_2 = C_2$ *(i.e.* $C_g \preceq C_1$ & $C_g \preceq C_2$*).*

Lisp-data SCP	*Refal SCP*
anti-unification problem aka most specific generalization solved by well-known algorithm since (Plotkin, 1970)	why not msg???

Refal msg problem

Definition

C_g *is* ***the most specific generalization*** *of* C_1 *and* C_2*, iff*
$C_g \preceq C_1 \;\&\; C_g \preceq C_2$ *and*
$\forall C'_g (C'_g \preceq C_1 \;\&\; C'_g \preceq C_2 \Rightarrow C'_g \preceq C_g)$.

Refal msg problem

Definition

C_g *is **the most specific generalization** of* C_1 *and* C_2*, iff*
$C_g \preceq C_1$ & $C_g \preceq C_2$ *and*
$\forall C'_g (C'_g \preceq C_1 \,\&\, C'_g \preceq C_2 \Rightarrow C'_g \preceq C_g)$.

Fails: given $C_1 = A$, $C_2 = AA$, $C_{g_1} = A\,w$, $C_{g_2} = w\,A$, both C_{g_1} and C_{g_2} are generalizations, but there is no σ s.t. $C_{g_2}\sigma = C_{g_1} \vee C_{g_1}\sigma = C_{g_2}$.

Refal msg problem

Definition

C_g *is* ***the most specific generalization*** *of* C_1 *and* C_2*, iff* $C_g \preceq C_1$ & $C_g \preceq C_2$ *and* $\forall C'_g(C'_g \preceq C_1 \;\&\; C'_g \preceq C_2 \Rightarrow C'_g \preceq C_g)$.

Fails: given $C_1 = A$, $C_2 = AA$, $C_{g_1} = A\,w$, $C_{g_2} = w\,A$, both C_{g_1} and C_{g_2} are generalizations, but there is no σ s.t. $C_{g_2}\sigma = C_{g_1} \vee C_{g_1}\sigma = C_{g_2}$.

Cause: $\preceq$ is not closed *w.r.t.* the supremums on the set of the generalizations.

Refal msg problem

Definition

C_g *is a **least general generalization** of* C_1 *and* C_2*, iff*
$C_g \preceq C_1$ & $C_g \preceq C_2$ *and*
$\forall C'_g (C'_g \preceq C_1 \;\&\; C'_g \preceq C_2 \;\&\; C_g \preceq C'_g \Rightarrow C'_g \preceq C_g)$.

$C_g \preceq C'_g$ & $C'_g \preceq C_g \Rightarrow C_g$ is a renaming of C'_g???

Refal msg problem

Definition

$\mathrm{C_g}$ *is a* ***least general generalization*** *of* C_1 *and* C_2*, iff*
$\mathrm{C_g} \preceq \mathrm{C}_1$ & $\mathrm{C_g} \preceq \mathrm{C}_2$ *and*
$\forall \mathrm{C}'_\mathrm{g} (\mathrm{C}'_\mathrm{g} \preceq \mathrm{C}_1 \,\&\, \mathrm{C}'_\mathrm{g} \preceq \mathrm{C}_2 \,\&\, \mathrm{C_g} \preceq \mathrm{C}'_\mathrm{g} \Rightarrow \mathrm{C}'_\mathrm{g} \preceq \mathrm{C_g})$.

$\mathrm{C_g} \preceq \mathrm{C}'_\mathrm{g} \,\&\, \mathrm{C}'_\mathrm{g} \preceq \mathrm{C_g} \Rightarrow \mathrm{C_g}$ is a renaming of C'_g???

No! Given $\mathrm{C_g} = w_1 w_2 w_2 u$ and $\mathrm{C}'_\mathrm{g} = w' u'$, both $\mathrm{C_g} \preceq \mathrm{C}'_\mathrm{g}$ and $\mathrm{C}'_\mathrm{g} \preceq \mathrm{C_g}$.

Refal msg problem

Definition

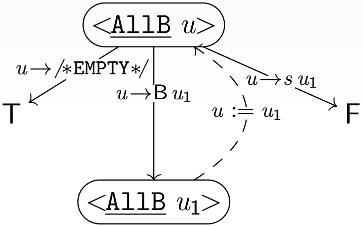
$\mathrm{C_g}$ *is a **least general generalization** of* $\mathrm{C_1}$ *and* $\mathrm{C_2}$*, iff*
$\mathrm{C_g} \preceq \mathrm{C_1}$ & $\mathrm{C_g} \preceq \mathrm{C_2}$ *and*
$\forall \mathrm{C'_g}(\mathrm{C'_g} \preceq \mathrm{C_1}$ & $\mathrm{C'_g} \preceq \mathrm{C_2}$ & $\mathrm{C_g} \preceq \mathrm{C'_g} \Rightarrow \mathrm{C'_g} \preceq \mathrm{C_g})$.

$\mathrm{C_g} \preceq \mathrm{C'_g}$ & $\mathrm{C'_g} \preceq \mathrm{C_g} \Rightarrow \mathrm{C_g}$ is a renaming of $\mathrm{C'_g}$???

No! Given $\mathrm{C_g} = w_1 w_2 w_2 u$ and $\mathrm{C'_g} = w' u'$, both $\mathrm{C_g} \preceq \mathrm{C'_g}$ and $\mathrm{C'_g} \preceq \mathrm{C_g}$.

Equivalence relation $\mathrm{C_1} \approx \mathrm{C_2}$ based on $\mathrm{C_1} \preceq \mathrm{C_2}$ & $\mathrm{C_2} \preceq \mathrm{C_1}$ is undefined.

Some more details



$s \neq$ B $\rightarrow$ the branches do not commute $\rightarrow$ problem of the negative information propagation.

Negative constraints may require non-constant-time checking: e.g. $w \neq$ e.1 A e.2.

General Refal SCP Problem

The finite-substitution language (while great for Prolog and lisp-data unification) is not appropriate to represent Refal data structures.

General Refal SCP Problem

The finite-substitution language (while great for Prolog and lisp-data unification) is not appropriate to represent Refal data structures.

Possible solution:

- word equations for representing driving results;
- pattern languages for comparing states in the process graph and representing negative constraints.

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

Word Equations at Work

(Abdulla et al, PLDI, 2017)

```
$ENTRY Go {
    (e.y) (e.x) (e.N) = <G0 (e.y) (e.x) (<Gram e.N>)>;}
Gram {
    'I' e.x = 'A' <Gram e.x> 'B';
    'S' e.x = <Gram e.x> 'B';
    /* EMPTY */ = /* EMPTY */;}
G0 {
    (e.y) (e.x) (e.w) = <Eq ('A' e.w e.x) (e.w e.y)>;}
Eq {
    (e.X) (e.X) = 'T';
    (e.X) (e.Y) = 'F';
}
```

If `'T'` is returned, then `e.N` must be `/* EMPTY */`, otherwise $\mathtt{Awx} = \mathtt{wy}$ has no solution.

Word Equations at Work

(Trinh et al, CAV, 2016)

```
$ENTRY Go {
    e.p = <G0 <GramA e.p> <GramB e.p> <GramC e.p>>; }
GramA {
    'I' e.x = 'A' <GramA e.x>;
    /* EMPTY */ = /* EMPTY */; }
GramB {
    'I' e.x = 'B' <GramB e.x>;
    /* EMPTY */ = /* EMPTY */; }
GramC {
    'I' e.x = 'C' <GramC e.x>;
    /* EMPTY */ = /* EMPTY */; }
G0 {  /* EMPTY */ = 'T';
    e.x 'D' = 'F';
    e.x t.y = <G0 e.x>; }
```

Generalization preserves information in the form of the word equations.

Flat Pattern Languages

Definition

Given an alphabet Σ *and a pattern* P*, the language* $\mathscr{L}(\mathrm{P})$ *recognized by* P *is a set of* $\Phi \in \Sigma^*$*, s.t.* σ*:* $\mathrm{P}\sigma = \Phi$*. The pattern* P_1 *is embedded in* P_2*, if* $\mathscr{L}(\mathrm{P}_1) \subseteq \mathscr{L}(\mathrm{P}_2)$.

If $\mathtt{e.x}\sigma = /*\,\mathtt{EMPTY}*/$ is allowed — erasing PL (EPL).
Otherwise — non-erasing PL (NePL).

- EPL recognized by the constant $\mathrm{P} \in \Sigma^*$ is $\{\mathrm{P}\}$.
- EPL recognized by $\mathrm{P} = \mathtt{e.x}_1\,\mathtt{e.x}_2 \ldots \mathtt{e.x}_n$ is Σ^*.

Pattern Languages at Work

`replace_all`-problem:

```
$ENTRY Go {
   e.q = <Check <ReplaceAB (/* EMPTY */)e.q>>;}
ReplaceAB {
   (/* EMPTY */)e.x 'AAB' e.y
             = <ReplaceAB (/* EMPTY */)e.x 'A' e.y>;
   (e.z)e.x 'AB' e.y = <ReplaceAB (e.z e.x) e.y>;
   (e.x)e.Other = e.x e.Other;
}
Check {
   e.x1 'AB' e.x2 = 'F';
   e.Z = 'T';
}
```

Solved using folding and checking negative constraints in terms of the pattern languages.

Pattern Languages at Work

One more `replace_all`-problem:

```
$ENTRY Go {
   e.p = <Check <DelAB (/* EMPTY */) e.p>>; }
DelAB {
   (/* EMPTY */) 'AB' e.x2 = <DelAB (/* EMPTY */) e.x2>;
   (e.x1 t.y1) 'AB' e.x2 = <DelAB (e.x1) t.y1 e.x2>;
   (e.x1) e.z t.y1 'AB' e.x2
                = <DelAB (e.x1 e.z) t.y1 e.x2>;
   (e.x1) e.x2 = e.x1 e.x2;}
Check {
   e.x1 'AB' e.x2 = 'F';
   e.Z = 'T';
}
```

Solved using folding and checking negative constraints in terms of the pattern languages.

Solving Word Equations via Supercompilation

(based on the VPT-2021 talk)

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 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 $\mathsf{AB}\mathtt{xxyy} = \mathtt{xxyy}\mathsf{BA}$ 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

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

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

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

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

Definition

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

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

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

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.

Narrs ::= (Narr) Narrs | ε

Narr ::= $'\mathrm{Var} \to \mathrm{Char}\ \mathrm{Var}' \,|\, '\mathrm{Var} \to \mathrm{Var}_1\,\mathrm{Var}' \,|\, '\mathrm{Var} \to \varepsilon'$

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

$'\mathtt{x} \to \varepsilon'$	$\mathtt{x}\,\Phi_1 = \Psi_1$ 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 $\mathrm{WI}_{\mathscr{L}}$ takes a finite sequence $(\sigma_1)(\sigma_2)\dots(\sigma_n)$ and a datum $\langle \Phi_i = \Psi_i \rangle_{i=1}^{m}$.

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

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

- 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 $\mathrm{WI}_{\mathscr{L}}$ 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 $\mathcal{E}qs$.

Residual programs: restricted Refal

Function definition

Definition ::= Name { Rule$^+$ }

Rule ::= Pattern = Expression;

Pattern ::= (Narr) | (Narr) ++ Pattern | `e.P` | ε

Expression ::= T | F | <Name `e.P`>

There `e.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 `e.P`.

Examples

Given the equation $\mathrm{A}\mathtt{x} = \mathtt{x}\mathrm{A}$, the supercompiler produces the following residual program, where the entry point is <F e.P>, and e.P is a variable ranging over Narrs.

```
F {
    ('x → ε') = T;
    ('x → Ax') ++ e.P = <F e.P>;
    e.P = F; }
```

Given the equation $\mathrm{A}\mathtt{x} = \mathtt{x}\mathrm{B}$, the residual program is as follows, where the entry point is <G e.P>.

```
G {
    ('x → ε') = F;
    ('x → Ax') ++ e.P = <G e.P>;
    e.P = F; }
```

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 $\mathrm{WIBase}_{\mathscr{L}}$

Structure of `Smpl` function

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

Further we refer to this simplification operation as Reduce.

Input format

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

$$\mathtt{Go}\ \{\mathtt{e.P} = <\mathtt{Main}\ (\mathtt{e.P})\ \texttt{++}\ <\mathtt{Smpl}\ (\ )\ \texttt{++}\ \mathcal{E}qs>>;\ \}$$

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

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

Structure of `Smpl` function

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

Input format

$$\texttt{Go}\ \{\mathrm{e.P} = <\texttt{Main}\ (\mathrm{e.P}) \mathbin{++} <\texttt{Smpl}\ 0 \mathbin{++} (\ ) \mathbin{++} \mathcal{E}qs>>;\ \}$$

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

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

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

Structure of `Smpl` function

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

Input format

$$\mathtt{Go}\ \{\mathtt{e.P} = \mathtt{<Main}\ (\mathtt{e.P})\ \text{++}\ \mathtt{<Smpl}\ 0\ \text{++}\ (\ )\ \text{++}\ \mathcal{E}qs\mathtt{>>};\ \}$$

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

Optimality lemma

Lemma

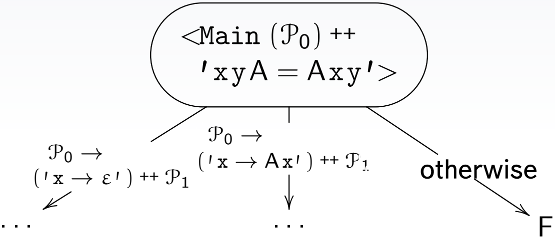
All the folding operations in the process graph of $\mathsf{WI}_{\mathscr{L}}(\mathcal{P}, \langle \Phi_i = \Psi_i \rangle_{i=1}^{n})$ occur only on the pairs of the configurations:

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

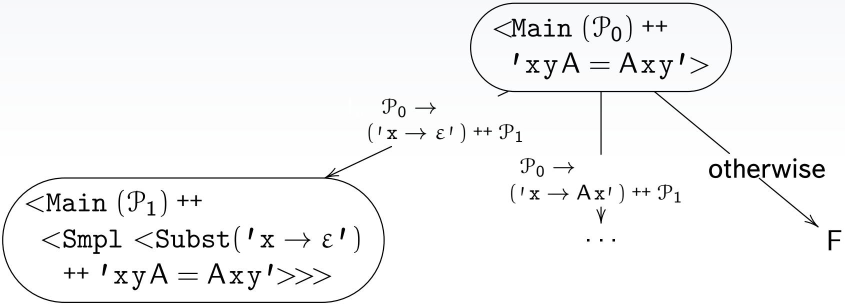
where $\mathcal{P}_j$ is a parameter, and the equation system does not contain parameters.

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 $\langle \Phi_i = \Psi_i \rangle_{i=1}^{n}$.

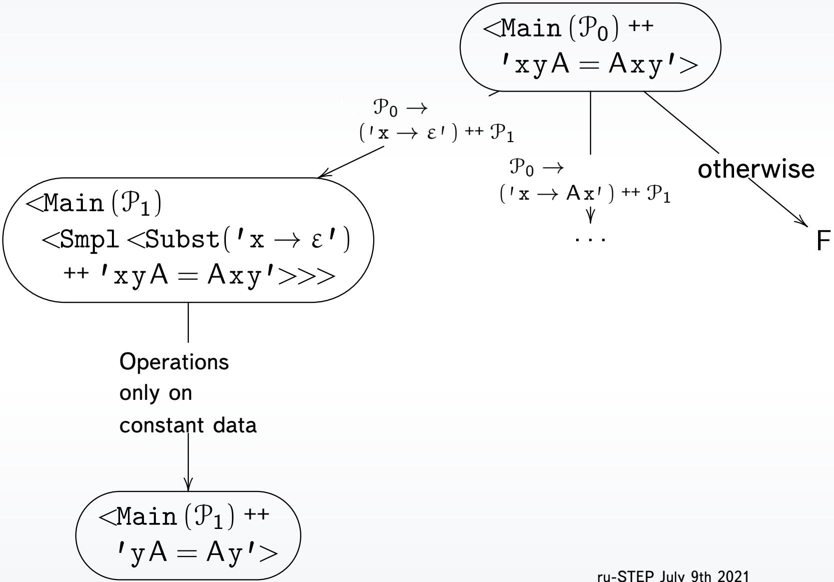
Generating the narrowings



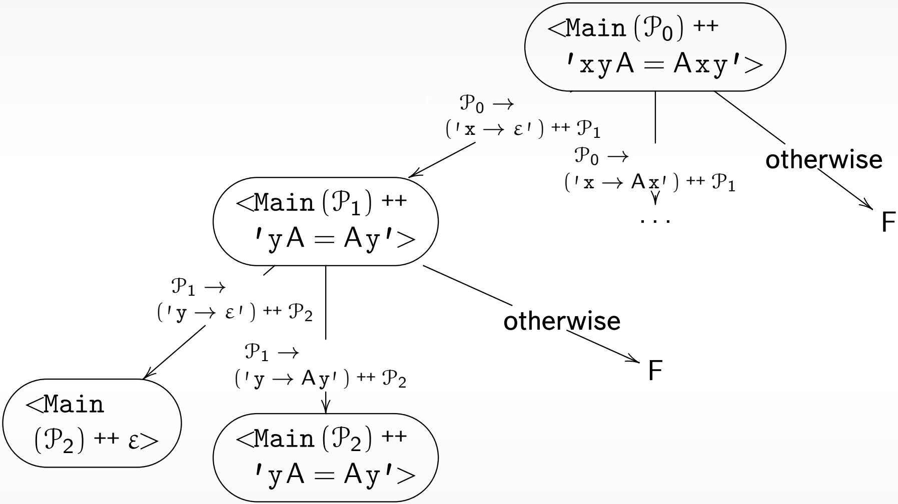
Generating the new configuration



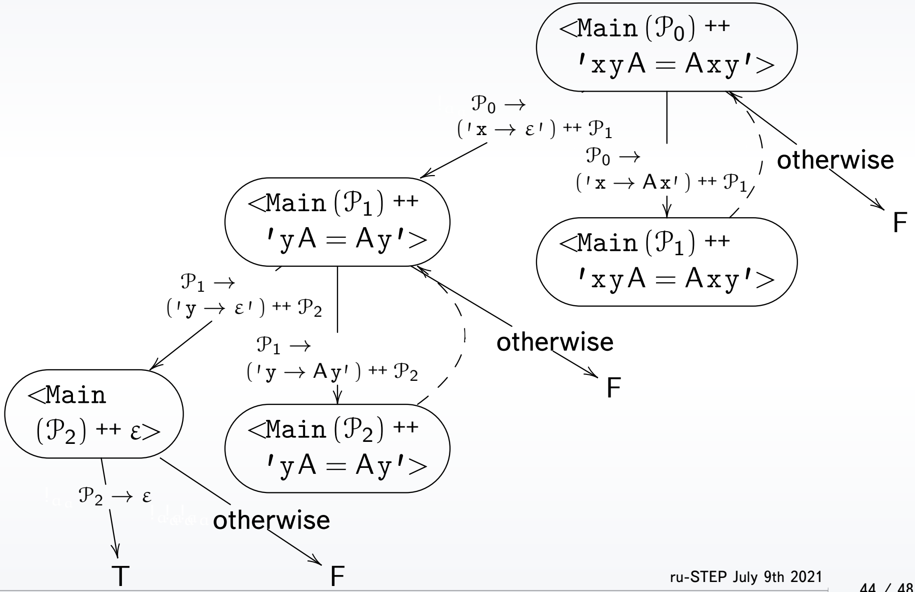
Transient operations



The next unfolding step

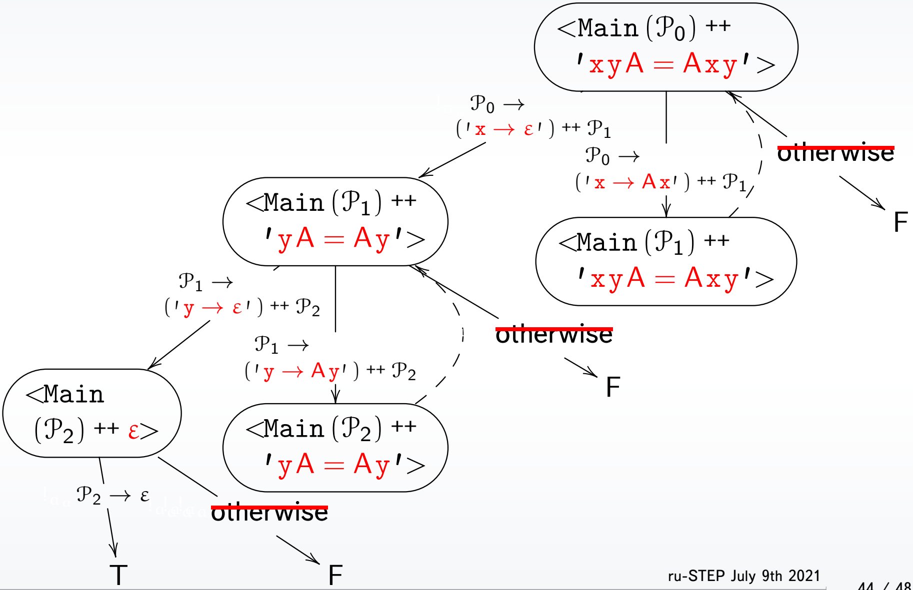


The folding

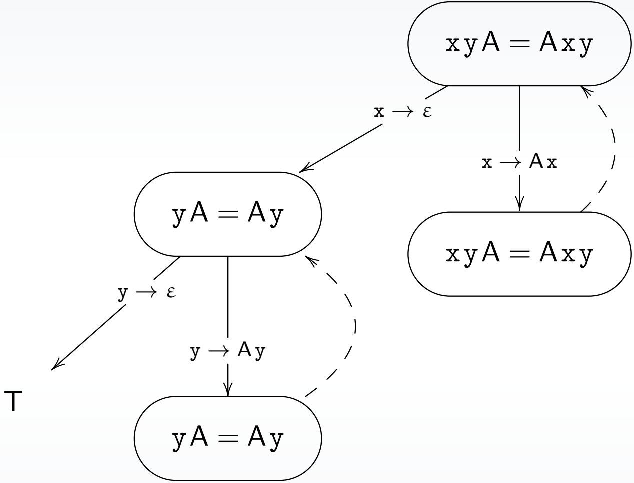


<Main ($\mathcal{P}_0$) ++ 'xyA = Axy'>
$\mathcal{P}_0 \to$ ('x → ε') ++ $\mathcal{P}_1$
$\mathcal{P}_0 \to$ ('x → Ax') ++ $\mathcal{P}_1$
otherwise
F
<Main ($\mathcal{P}_1$) ++ 'yA = Ay'>
<Main ($\mathcal{P}_1$) ++ 'xyA = Axy'>
$\mathcal{P}_1 \to$ ('y → ε') ++ $\mathcal{P}_2$
$\mathcal{P}_1 \to$ ('y → 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.* $\mathtt{x}_1\mathtt{x}_2\mathtt{x}_3\mathrm{ABABAB} = \mathrm{AAABBB}\,\mathtt{x}_2\mathtt{x}_3\mathtt{x}_1$);
- the regular-ordered equations with var-repetitions and no solution (*e.g.* $\mathrm{AB}\mathtt{xxyy} = \mathtt{xxyy}\mathrm{BA}$).

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.

Why do we lose in time?

We solve the two different tasks:

- the solvers are finding at least one solution;
- we are finding the description of all solutions.

Implications

- coefficient-free equations are always solved by the solvers;
- our method is very slow on the straight-line equations.

Conclusions

- Refal syntax is convenient for modelling string manipulating systems.
- Using the Refal-friendly structures in a supercompiler allows it to manage many string-verifying tasks.
- The two main problems:
  - complexity of the algorithms over the Refal-friendly data structures;
  - the exotic syntax is scaring for most people who does not know Refal.

Thank you for your attention!