On Unfolding for Programs Using Strings as a Data Type

A. P. Nemytykh

Program Systems Institute
Russian Academy of Sciences

Workshop on Verification and Program Transformation
Vienna, 2014

Unfolding

- As a rule, program transformation methods based on operational semantics unfold a semantic tree of a given program.
- Unfolding is one of the basic operations, which is a meta-extension of one step of the abstract machine executing the program.
- Widely used in program analysis, including verification.

Unfolding

- As a rule, program transformation methods based on operational semantics unfold a semantic tree of a given program.
- Unfolding is one of the basic operations, which is a meta-extension of one step of the abstract machine executing the program.
- Widely used in program analysis, including verification.

Unfolding

- As a rule, program transformation methods based on operational semantics unfold a semantic tree of a given program.
- Unfolding is one of the basic operations, which is a meta-extension of one step of the abstract machine executing the program.
- Widely used in program analysis, including verification.

The Rooted Tree

Nature

The Rooted Binary Tree

LISP Data

The Forest

Nature

The Forest

REFAL Data

Arbitrary trees. Associative concatenation.

The Problem

Nature

The Problem

Programming

The Problem

Simplification

Associative concatenation.

Strings as a Data Type

Strings form a fundamental data type used in most programming languages.

- Prolog-III includes string unification.
- The functional programming language Refal is based on associative concatenation constructor.
- Constraint logic programs manipulating the string values are used in industrial programming.

Strings as a Data Type

Related Works

Recently, a number of techniques producing string constraints have been suggested for:

- program verification: D. Shannon et al (2007);
- automatic testing:
  P. Godefroid et al (2008), N. Bjørner et al (2009);
- . . . . . .
- constraint logic programs manipulating the string values are used in industrial programming.

String-constraint solvers are used in many testing and analysis tools:

- 2003: A. S. Christensen et al;
- 2007: M. Emmi et al; G. Wassermann and Z. Su; X. Fu et al;
- 2008: H. Ruan et al;

Strings as a Data Type

Related Works

Recently, a number of techniques producing string constraints have been suggested for:

- program verification: D. Shannon et al (2007);
- automatic testing:
  P. Godefroid et al (2008), N. Bjørner et al (2009);
- ...
- constraint logic programs manipulating the string values are used in industrial programming.

String-constraint solvers are used in many testing and analysis tools:

- 2003: A. S. Christensen et al;
- 2007: M. Emmi et al; G. Wassermann and Z. Su; X. Fu et al;
- 2008: H. Ruan et al;

Strings as a Data Type

Related Works

Recently, a number of techniques producing string constraints have been suggested for:

- program verification: D. Shannon et al (2007);
- automatic testing:
  P. Godefroid et al (2008), N. Bjørner et al (2009);
- . . . . . .
- constraint logic programs manipulating the string values are used in industrial programming.

String-constraint solvers are used in many testing and analysis tools:

- 2003: A. S. Christensen et al;
- 2007: M. Emmi et al; G. Wassermann and Z. Su; X. Fu et al;
- 2008: H. Ruan et al;

Strings as a Data Type

Related Works

Recently, a number of techniques producing string constraints have been suggested for:

- program verification: D. Shannon et al (2007);
- automatic testing:
  P. Godefroid et al (2008), N. Bjørner et al (2009);
- . . . . . .
- constraint logic programs manipulating the string values are used in industrial programming.

String-constraint solvers are used in many testing and analysis tools:

- 2003: A. S. Christensen et al;
- 2007: M. Emmi et al; G. Wassermann and Z. Su; X. Fu et al;
- 2008: H. Ruan et al;

Strings as a Data Type

Related Works

Recently, a number of techniques producing string constraints have been suggested for:

- program verification: D. Shannon et al (2007);
- automatic testing:
  P. Godefroid et al (2008), N. Bjørner et al (2009);
- . . . . . .
- constraint logic programs manipulating the string values are used in industrial programming.

String-constraint solvers are used in many testing and analysis tools:

- 2003: A. S. Christensen et al;
- 2007: M. Emmi et al; G. Wassermann and Z. Su; X. Fu et al;
- 2008: H. Ruan et al;

Unfolding

Related Works

Algorithms unfolding programs were being intensively studied for a long time:

- V. F. Turchin (in Russian: 1968, . . .), (in English: 1980, . . .);
- N. D. Jones (1994, . . .);
- A. Pettorossi, M. Proietti (1997, . . .);
- M. Gabbrielli et al. (2013);
- and so on . . .

Unfolding

Related Works

Algorithms unfolding programs were being intensively studied for a long time:

- V. F. Turchin (in Russian: 1968, . . .), (in English: 1980, . . .);
- N. D. Jones (1994, . . .);
- A. Pettorossi, M. Proietti (1997, . . .);
- M. Gabbrielli et al. (2013);
- and so on . . .

Unfolding

Related Works

Algorithms unfolding programs were being intensively studied for a long time:

- V. F. Turchin (in Russian: 1968, . . .), (in English: 1980, . . .);
- N. D. Jones (1994, . . .);
- A. Pettorossi, M. Proietti (1997, . . .);
- M. Gabbrielli et al. (2013);
- and so on . . .

Unfolding

Related Works

Algorithms unfolding programs were being intensively studied for a long time:

- V. F. Turchin (in Russian: 1968, . . .), (in English: 1980, . . .);
- N. D. Jones (1994, . . .);
- A. Pettorossi, M. Proietti (1997, . . .);
- M. Gabbrielli et al. (2013);
- and so on . . .

Unfolding

Related Works

Algorithms unfolding programs were being intensively studied for a long time:

- V. F. Turchin (in Russian: 1968, . . .), (in English: 1980, . . .);
- N. D. Jones (1994, . . .);
- A. Pettorossi, M. Proietti (1997, . . .);
- M. Gabbrielli et al. (2013);
- and so on . . .

Unfolding

Related Works

The associative concatenation was stood by the wayside of the stream and mainly was considered only in the context of the Refal language:

- V. F. Turchin (in Russian: 1968, . . .), (in English: 1980, . . .);

The Problem

Associative Concatenation

Ambiguous pattern matching:

'AB'		'CD'		'D'
'AB'	'CDCDCD'	'CD'	'D'	'D'
'AB'	'CDCD'	'CD'	'CDD'	'D'
'AB'	'CD'	'CD'	'CDCDD'	'D'
'AB'	''	'CD'	'CDCDCDD'	'D'

- The time taken by the pattern-matching is not uniformly bounded above.

The Problem

Associative Concatenation

Ambiguous pattern matching:

'AB'	X	'CD'	Y	'D'
'AB'	'CDCDCD'	'CD'	'D'	'D'
'AB'	'CDCD'	'CD'	'CDD'	'D'
'AB'	'CD'	'CD'	'CDCDD'	'D'
'AB'	''	'CD'	'CDCDCDD'	'D'

- The time taken by the pattern-matching is not uniformly bounded above
  - by the length of the input string.

The Problem

Associative Concatenation

Ambiguous pattern matching:

'AB'		'CD'		'D'
'AB'	'CDCDCD'	'CD'	'D'	'D'
'AB'	'CDCD'	'CD'	'CDD'	'D'
'AB'	'CD'	'CD'	'CDCDD'	'D'
'AB'	''	'CD'	'CDCDCDD'	'D'

- The time taken by the pattern-matching is not uniformly bounded above
  - by the length of the input string.

The Pattern-Matching and Words Equations

Associative Concatenation

Such a pattern-matching connects the program interpretation with solving a class of word equations:

$$'\mathtt{AB}'\,\boxed{X}\,'\mathtt{CD}'\,\boxed{Y}\,'\mathtt{D}' = '\mathtt{ABCDCDCDCDDD}'$$

- The right sides of the equations do not contain variables.

The Pattern-Matching and Words Equations

Associative Concatenation

Such a pattern-matching connects the program interpretation with solving a class of word equations:

- The right-sides of the equations do not contain variables.

Words Equations

Definition

Given a finite alphabet Σ of constants and a set of variables X disjoint with Σ, a word expression is $L = R$, where $L, R \in (\Sigma \cup X)^*$.

Definition

A solution of a word equation $L = R$ is any substitution of its unknowns in the word equation by words from Σ^* that turns the equation $L = R$ into literal/syntactic equality.

Words Equations

Definition

Given a finite alphabet Σ of constants and a set of variables X disjoint with Σ, a word expression is $L = R$, where $L, R \in (\Sigma \cup X)^*$.

Definition

A solution of a word equation $L = R$ is any substitution of its unknowns in the word equation by words from Σ^* that turns the equation $L = R$ into literal/syntactic equality.

- Unfolding based on the extended patter-matching connects the program transformation with solving the general word equations.

Simplest Examples

Example

$'\mathtt{a}'\ X = X\ '\mathtt{a}'$

- $X := '\mathtt{a}'$ is a solution;
- $X := '\mathtt{aaa}'$ is a solution;
- ...

Example

$X\ '\mathtt{a}'\ Y = P\ '\mathtt{a}'\ Q$

WordEquation Problem vs. Hilbert's 10th Problem

WordEquation Problem

The problem of deciding whether a given word equation has a solution is an instance of Hilbert's 10th Problem.

Undecidable problems.

⋮

In 1970 Matiyasevich: Hilbert's 10th Problem is undecidable.

●●●●●●●●●●●●●●●●●● The Borderline ●●●●●●●●●●●●●●●●●●

In 1977 Makanin: WordEquation is decidable!

⋮

Decidable problems.

WordEquation Problem vs. Hilbert's 10th Problem

WordEquation Problem

The problem of deciding whether a given word equation has a solution is an instance of Hilbert's 10th Problem.

Undecidable problems.

⋮

In 1970 Matiyasevich: Hilbert's 10th Problem is undecidable.

●●●●●●●●●●●●●●●●●● The Borderline ●●●●●●●●●●●●●●●●●●

In 1977 Makanin: WordEquation is decidable!

⋮

Decidable problems.

WordEquation Problem vs. Hilbert's 10th Problem

WordEquation Problem

The problem of deciding whether a given word equation has a solution is an instance of Hilbert's 10th Problem.

Undecidable problems.

⋮

In 1970 Matiyasevich: Hilbert's 10th Problem is undecidable.

●●●●●●●●●●●●●●●● The Borderline ●●●●●●●●●●●●●●●●

In 1977 Makanin: WordEquation is decidable!

⋮

Decidable problems.

The Pattern-Matching and Words Equations

Open Problem

Given a word equation, the problem is to describe the set of all its solutions.

Our Contribution

- a syntactic description of a class $\mathcal{C}$ of term rewriting systems manipulating with strings;
  - the programming language defined by $\mathcal{C}$ is Turing-complete;
- a new algorithm for one-step unfolding of the programs from $\mathcal{C}$;
- the pattern language of $\mathcal{C}$ in turn fixes a class $\mathcal{E}$ of word equations:
  - given an equation from $\mathcal{E}$, a nondeterministic version of the algorithm results in a description (by a term rewriting system) of the corresponding solution set.

Our Contribution

- a syntactic description of a class $\mathcal{C}$ of term rewriting systems manipulating with strings;
  - the programming language defined by $\mathcal{C}$ is Turing-complete;
- a new algorithm for one-step unfolding of the programs from $\mathcal{C}$;
- the pattern language of $\mathcal{C}$ in turn fixes a class $\mathcal{E}$ of word equations:
  - given an equation from $\mathcal{E}$, a nondeterministic version of the algorithm results in a description (by a term rewriting system) of the corresponding solution set.

Our Contribution

- a syntactic description of a class $\mathcal{C}$ of term rewriting systems manipulating with strings;
  - the programming language defined by $\mathcal{C}$ is Turing-complete;
- a new algorithm for one-step unfolding of the programs from $\mathcal{C}$;
- the pattern language of $\mathcal{C}$ in turn fixes a class $\mathcal{E}$ of word equations:
  - given an equation from $\mathcal{E}$, a nondeterministic version of the algorithm results in a description (by a term rewriting system) of the corresponding solution set.

Our Contribution

- a syntactic description of a class $\mathcal{C}$ of term rewriting systems manipulating with strings;
  - the programming language defined by $\mathcal{C}$ is Turing-complete;
- a new algorithm for one-step unfolding of the programs from $\mathcal{C}$;
- the pattern language of $\mathcal{C}$ in turn fixes a class $\mathcal{E}$ of word equations:
  - given an equation from $\mathcal{E}$, a nondeterministic version of the algorithm results in a description (by a term rewriting system) of the corresponding solution set.

Our Contribution

- a syntactic description of a class $\mathcal{C}$ of term rewriting systems manipulating with strings;
  - the programming language defined by $\mathcal{C}$ is Turing-complete;
- a new algorithm for one-step unfolding of the programs from $\mathcal{C}$;
- the pattern language of $\mathcal{C}$ in turn fixes a class $\mathcal{E}$ of word equations:
  - given an equation from $\mathcal{E}$, a nondeterministic version of the algorithm results in a description (by a term rewriting system) of the corresponding solution set.

Term Rewriting Systems Based on Pattern Matching

Example Program

The data set is finite strings, including the empty string λ; the concatenation is associative.

$\langle t, R \rangle$ checks whether two given input data (parameters) are equal

t is **eq(#e.p,#e.q)** and R is:

```
eq(s.x ++ e.xs,s.x ++ e.ys) = eq(e.xs,e.ys);
eq(λ,λ) = 'T';
eq(e.xs,e.ys) = 'F';
```

- Two kinds of variables:
  - s-variables: range over characters
  - e-variables: range over the whole set of the strings
- Two kinds of parameters: s- and e-parameters.

'b'++'a' $\equiv$ **'ba';** **'aba'++e.x** $\equiv$ **'a':'b':'a':e.x**

Term Rewriting Systems Based on Pattern Matching

Example Program

The data set is finite strings, including the empty string λ; the concatenation is associative.

$\langle t, R \rangle$ checks whether two given input data (parameters) are equal

t is **eq(#e.p,#e.q)** and R is:

```
eq(s.x ++ e.xs,s.x ++ e.ys) = eq(e.xs,e.ys);
eq(λ,λ) = 'T';
eq(e.xs,e.ys) = 'F';
```

- Two kinds of variables:
  - s.-variables: range over characters;
  - e.-variables: range over the whole set of the strings;
- Two kinds of parameters: s.- and e.-parameters.

'b'++'a' $\equiv$ **'ba';** **'aba'++e.x** $\equiv$ **'a':'b':'a':e.x**

Term Rewriting Systems Based on Pattern Matching

Example Program

The data set is finite strings, including the empty string λ; the concatenation is associative.

$\langle t, R \rangle$ checks whether two given input data (parameters) are equal

t is **eq(#e.p,#e.q)** and R is:

```
eq(s.x ++ e.xs,s.x ++ e.ys) = eq(e.xs,e.ys);
eq(λ,λ) = 'T';
eq(e.xs,e.ys) = 'F';
```

- Two kinds of variables:
  - s.-variables: range over characters;
  - e.-variables: range over the whole set of the strings;
- Two kinds of parameters: s.- and e.-parameters.

'b'++'a' $\equiv$ **'ba';** **'aba'++e.x** $\equiv$ **'a':'b':'a':e.x**

Term Rewriting Systems Based on Pattern Matching

Example Program

The data set is finite strings, including the empty string λ; the concatenation is associative.

$\langle t, R \rangle$ checks whether two given input data (parameters) are equal

t is `eq(#e.p,#e.q)` and R is:

```
eq(s.x ++ e.xs,s.x ++ e.ys) = eq(e.xs,e.ys);
eq(λ,λ) = 'T';
eq(e.xs,e.ys) = 'F';
```

- Two kinds of variables:
  - s.-variables: range over characters;
  - e.-variables: range over the whole set of the strings;
- Two kinds of parameters: s.- and e.-parameters.

`'b'++'a'` $\equiv$ `'ba';` `'aba'++e.x` $\equiv$ `'a':'b':'a':e.x`

Term Rewriting Systems Based on Pattern Matching

Example Program

The data set is finite strings, including the empty string λ; the concatenation is associative.

$\langle t, R \rangle$ checks whether two given input data (parameters) are equal

t is `eq(#e.p,#e.q)` and R is:

```
eq(s.x ++ e.xs,s.x ++ e.ys) = eq(e.xs,e.ys);
eq(λ,λ) = 'T';
eq(e.xs,e.ys) = 'F';
```

- Two kinds of variables:
  - s.-variables: range over characters;
  - e.-variables: range over the whole set of the strings;
- Two kinds of parameters: s.- and e.-parameters.

`'b'++'a'` $\equiv$ `'ba';` `'aba'++e.x` $\equiv$ `'a':'b':'a':e.x`

Term Rewriting Systems Based on Pattern Matching

Example Program

The data set is finite strings, including the empty string λ; the concatenation is associative.

$\langle t, R \rangle$ checks whether two given input data (parameters) are equal

t is `eq(#e.p,#e.q)` and R is:

```
eq(s.x ++ e.xs,s.x ++ e.ys) = eq(e.xs,e.ys);
eq(λ,λ) = 'T';
eq(e.xs,e.ys) = 'F';
```

- Two kinds of variables:
  - s.-variables: range over characters;
  - e.-variables: range over the whole set of the strings;
- Two kinds of parameters: s.- and e.-parameters.

`'b'++'a'` $\equiv$ `'ba';` `'aba'++e.x` $\equiv$ `'a':'b':'a':e.x`

Term Rewriting Systems Based on Pattern Matching

Example Program

The data set is finite strings, including the empty string λ; the concatenation is associative.

$\langle t, R \rangle$ checks whether two given input data (parameters) are equal

t is `eq(#e.p,#e.q)` and R is:

```
eq(s.x ++ e.xs,s.x ++ e.ys) = eq(e.xs,e.ys);
eq(λ,λ) = 'T';
eq(e.xs,e.ys) = 'F';
```

- Two kinds of variables:
  - s.-variables: range over characters;
  - e.-variables: range over the whole set of the strings;
- Two kinds of parameters: s.- and e.-parameters.

`'b'++'a'` $\equiv$ `'ba';` `'aba'++e.x` $\equiv$ `'a':'b':'a':e.x`

Term Rewriting Systems Based on Pattern Matching

Example Program

Two variants of a pseudocode for functional programs: $\mathcal{L}$ is deterministic, $\mathcal{L}_*$ is nondeterministic.

Program $\langle t, R \rangle$ checks whether two given input data are equal

t is `eq(#e.p,#e.q)` and R is:

```
eq(s.x ++ e.xs,s.x ++ e.ys) = eq(e.xs,e.ys);
eq(λ,λ) = 'T';
eq(e.xs,e.ys) = 'F';
```

- $\mathcal{L}$: the sentences are ordered to be matched.
- $\mathcal{L}_*$: the sentences are not ordered.

Term Rewriting Systems Based on Pattern Matching

Example Program

Two variants of a pseudocode for functional programs: $\mathcal{L}$ is deterministic, $\mathcal{L}_*$ is nondeterministic.

Program $\langle t, R \rangle$ checks whether two given input data are equal

t is `eq(#e.p,#e.q)` and R is:

```
eq(s.x ++ e.xs,s.x ++ e.ys) = eq(e.xs,e.ys);
eq(λ,λ) = 'T';
eq(e.xs,e.ys) = 'F';
```

- $\mathcal{L}$: the sentences are ordered to be matched.
- $\mathcal{L}_*$: the sentences are not ordered.

Term Rewriting Systems Based on Pattern Matching

Example Program

Two variants of a pseudocode for functional programs: $\mathcal{L}$ is deterministic, $\mathcal{L}_*$ is nondeterministic.

Program $\langle t, R \rangle$ checks whether two given input data are equal

t is `eq(#e.p,#e.q)` and R is:

```
eq(s.x ++ e.xs,s.x ++ e.ys) = eq(e.xs,e.ys);
eq(λ,λ) = 'T';
eq(e.xs,e.ys) = 'F';
```

- $\mathcal{L}$: the sentences are ordered to be matched.
- $\mathcal{L}_*$: the sentences are not ordered.

Term Rewriting Systems Based on Pattern Matching

Example Program

Two variants of a pseudocode for functional programs: $\mathcal{L}$ is deterministic, $\mathcal{L}_*$ is nondeterministic.

Program $\langle t, R \rangle$ checks whether two given input data are equal

t is `eq(#e.p,#e.q)` and R is:

```
eq(s.x ++ e.xs,s.x ++ e.ys) = eq(e.xs,e.ys);
eq(λ,λ) = 'T';
eq(e.xs,e.ys) = 'F';
```

- $\mathcal{L}$: the sentences are ordered to be matched.
- $\mathcal{L}_*$: the sentences are not ordered.

Term Rewriting Systems Based on Pattern Matching

Example Program

Two variants of a pseudocode for functional programs: $\mathcal{L}$ is deterministic, $\mathcal{L}_*$ is nondeterministic.

Program $\langle t, R \rangle$ checks whether two given input data are equal

t is `eq(#e.p,#e.q)` and R is:

```
eq(s.x ++ e.xs,s.x ++ e.ys) = eq(e.xs,e.ys);
eq(λ,λ) = 'T';
eq(e.xs,e.ys) = 'F';
```

- $\mathcal{L}$: the sentences are ordered to be matched.
- $\mathcal{L}_*$: the sentences are not ordered.

On Semantics of the Pattern Matching

$\mathcal{L}$: Markov's Rule

$$\text{pattern} = \text{string}_0$$

The e-variables are ordered according to theirs first occurrences in the pattern: $\texttt{'a'++e.x}_1\texttt{++'b'++e.y}_2\texttt{++e.x}_1\texttt{++'c'++e.z}_3$

Solutions:

$\sigma_1(\texttt{e.x}_1) = \texttt{str}_{11}, \sigma_1(\texttt{e.y}_2) = \texttt{str}_{21}, \sigma_1(\texttt{e.z}_3) = \texttt{str}_{31}$

$\sigma_2(\texttt{e.x}_1) = \texttt{str}_{12}, \sigma_2(\texttt{e.y}_2) = \texttt{str}_{22}, \sigma_2(\texttt{e.z}_3) = \texttt{str}_{32}$

.....

Deterministic choice:

1. the least e-variable takes the shortest value by σ_{i_0};
2. if such a choice still gives more than one solution, then apply the case (1) to $\sigma_{i_0}(\text{pattern}) = \text{string}_0$;

On Semantics of the Pattern Matching

$\mathcal{L}$: Markov's Rule

$$\text{pattern} = \text{string}_0$$

The e-variables are ordered according to theirs first occurrences in the pattern: $\texttt{'a'++e.x}_1\texttt{++'b'++e.y}_2\texttt{++e.x}_1\texttt{++'c'++e.z}_3$

Solutions:

$\sigma_1(\texttt{e.x}_1) = \texttt{str}_{11}, \sigma_1(\texttt{e.y}_2) = \texttt{str}_{21}, \sigma_1(\texttt{e.z}_3) = \texttt{str}_{31}$

$\sigma_2(\texttt{e.x}_1) = \texttt{str}_{12}, \sigma_2(\texttt{e.y}_2) = \texttt{str}_{22}, \sigma_2(\texttt{e.z}_3) = \texttt{str}_{32}$

.....

Deterministic choice:

1. the least e-variable takes the shortest value by σ_{i_0};
2. if such a choice still gives more than one solution, then apply the case (1) to $\sigma_{i_0}(\text{pattern}) = \text{string}_0$;

On Semantics of the Pattern Matching

$\mathcal{L}$: Markov's Rule

$$\text{pattern} = \text{string}_0$$

The e-variables are ordered according to theirs first occurrences in the pattern: $\texttt{'a'++e.x}_1\texttt{++'b'++e.y}_2\texttt{++e.x}_1\texttt{++'c'++e.z}_3$

Solutions:

$\sigma_1(\texttt{e.x}_1) = \texttt{str}_{11}, \sigma_1(\texttt{e.y}_2) = \texttt{str}_{21}, \sigma_1(\texttt{e.z}_3) = \texttt{str}_{31}$

$\sigma_2(\texttt{e.x}_1) = \texttt{str}_{12}, \sigma_2(\texttt{e.y}_2) = \texttt{str}_{22}, \sigma_2(\texttt{e.z}_3) = \texttt{str}_{32}$

.....

Deterministic choice:

1. the least e-variable takes the shortest value by σ_{i_0};
2. if such a choice still gives more than one solution, then apply the case (1) to $\sigma_{i_0}(\text{pattern}) = \text{string}_0$;

On Semantics of the Pattern Matching

$\mathcal{L}$: Markov's Rule

$$\text{pattern} = \text{string}_0$$

The e-variables are ordered according to theirs first occurrences in the pattern: $\texttt{'a'++e.x}_1\texttt{++'b'++e.y}_2\texttt{++e.x}_1\texttt{++'c'++e.z}_3$

Solutions:

$\sigma_1(\texttt{e.x}_1) = \texttt{str}_{11}, \sigma_1(\texttt{e.y}_2) = \texttt{str}_{21}, \sigma_1(\texttt{e.z}_3) = \texttt{str}_{31}$

$\sigma_2(\texttt{e.x}_1) = \texttt{str}_{12}, \sigma_2(\texttt{e.y}_2) = \texttt{str}_{22}, \sigma_2(\texttt{e.z}_3) = \texttt{str}_{32}$

.....

Deterministic choice:

1. the least e-variable takes the shortest value by σ_{i_0};
2. if such a choice still gives more than one solution, then apply the case (1) to $\sigma_{i_0}(\text{pattern}) = \text{string}_0$;

On Semantics of the Pattern Matching

$\mathcal{L}$: Markov's Rule

$$\text{pattern} = \text{string}_0$$

The e-variables are ordered according to theirs first occurrences in the pattern: $\texttt{'a'++e.x}_1\texttt{++'b'++e.y}_2\texttt{++e.x}_1\texttt{++'c'++e.z}_3$

Solutions:

$\sigma_1(\texttt{e.x}_1) = \texttt{str}_{11}, \sigma_1(\texttt{e.y}_2) = \texttt{str}_{21}, \sigma_1(\texttt{e.z}_3) = \texttt{str}_{31}$

$\sigma_2(\texttt{e.x}_1) = \texttt{str}_{12}, \sigma_2(\texttt{e.y}_2) = \texttt{str}_{22}, \sigma_2(\texttt{e.z}_3) = \texttt{str}_{32}$

.....

Deterministic choice:

1. the least e-variable takes the shortest value by σ_{i_0};
2. if such a choice still gives more than one solution, then apply the case (1) to $\sigma_{i_0}(\text{pattern}) = \text{string}_0$;

On Semantics of the Pattern Matching

$\mathcal{L}$: Markov's Rule

$$\text{pattern} = \text{string}_0$$

The e-variables are ordered according to theirs first occurrences in the pattern: $\texttt{'a'++e.x}_1\texttt{++'b'++e.y}_2\texttt{++e.x}_1\texttt{++'c'++e.z}_3$

Solutions:

$\sigma_1(\texttt{e.x}_1) = \texttt{str}_{11}, \sigma_1(\texttt{e.y}_2) = \texttt{str}_{21}, \sigma_1(\texttt{e.z}_3) = \texttt{str}_{31}$

$\sigma_2(\texttt{e.x}_1) = \texttt{str}_{12}, \sigma_2(\texttt{e.y}_2) = \texttt{str}_{22}, \sigma_2(\texttt{e.z}_3) = \texttt{str}_{32}$

.....

Deterministic choice:

1. the least e-variable takes the shortest value by σ_{i_0};
2. if such a choice still gives more than one solution, then apply the case (1) to $\sigma_{i_0}(\text{pattern}) = \text{string}_0$;

On Semantics of the Pattern Matching

$\mathcal{L}$: Markov's Rule

$$\text{pattern} = \text{string}_0$$

The e-variables are ordered according to theirs first occurrences in the pattern: $\texttt{'a'++e.x}_1\texttt{++'b'++e.y}_2\texttt{++e.x}_1\texttt{++'c'++e.z}_3$

Solutions:

$\sigma_1(\texttt{e.x}_1) = \texttt{str}_{11}, \sigma_1(\texttt{e.y}_2) = \texttt{str}_{21}, \sigma_1(\texttt{e.z}_3) = \texttt{str}_{31}$

$\sigma_2(\texttt{e.x}_1) = \texttt{str}_{12}, \sigma_2(\texttt{e.y}_2) = \texttt{str}_{22}, \sigma_2(\texttt{e.z}_3) = \texttt{str}_{32}$

.....

Deterministic choice:

1. the least e-variable takes the shortest value by σ_{i_0};
2. if such a choice still gives more than one solution, then apply the case (1) to $\sigma_{i_0}(\text{pattern}) = \text{string}_0$;

On Semantics of the Pattern Matching

$\mathcal{L}$: Markov's Rule

$$\texttt{'AB'++e.X++'CD'++e.Y++'D'} = \texttt{'ABCDCDCDCDDD'}$$

Deterministic choice:

'AB'		'CD'		'D'
'AB'	'CDCDCD'	'CD'	'D'	'D'
'AB'	'CDCD'	'CD'	'CDD'	'D'
'AB'	'CD'	'CD'	'CDCDD'	'D'
'AB'	''	'CD'	'CDCDCDD'	'D'

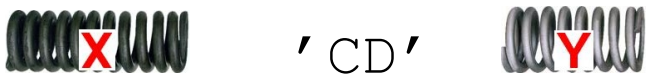
- $\sigma_4(\texttt{e.X}) = \lambda$, $\sigma_4(\texttt{e.Y}) = \texttt{'CDCDCDD'}$

On Semantics of the Pattern Matching

The Language $\mathcal{L}_*$

```
'AB'++e.X++'CD'++e.Y++'D' = 'ABCDCDCDCDDD'
```

Nondeterministic choice:




```
'AB'    X     'CD'    Y       'D'
'AB'  'CDCDCD'  'CD'  'D'      'D'
'AB'  'CDCD'    'CD'  'CDD'    'D'
'AB'  'CD'      'CD'  'CDCDD'  'D'
'AB'  ''        'CD'  'CDCDCDD' 'D'
```

On Semantics of the Pattern Matching

The Language $\mathcal{L}_*$

$$'AB' ++ e.X ++ 'CD' ++ e.Y ++ 'D' = 'ABCDCDCDCDDD'$$

Nondeterministic choice:

'AB'		'CD'		'D'
'AB'	'CDCDCD'	'CD'	'D'	'D'
'AB'	'CDCD'	'CD'	'CDD'	'D'
'AB'	'CD'	'CD'	'CDCDD'	'D'
'AB'	''	'CD'	'CDCDCDD'	'D'

On Semantics of the Pattern Matching

The Language $\mathcal{L}_*$

$$\texttt{'AB'++e.X++'CD'++e.Y++'D'} = \texttt{'ABCDCDCDCDDD'}$$

Nondeterministic choice:

'AB'		'CD'		'D'
'AB'	'CDCDCD'	'CD'	'D'	'D'
'AB'	'CDCD'	'CD'	'CDD'	'D'
'AB'	'CD'	'CD'	'CDCDD'	'D'
'AB'	''	'CD'	'CDCDCDD'	'D'

On Semantics of the Pattern Matching

The Language $\mathcal{L}_*$

```
'AB'++e.X++'CD'++e.Y++'D' = 'ABCDCDCDCDDD'
```

Nondeterministic choice:

'AB'		'CD'		'D'
'AB'	'CDCDCD'	'CD'	'D'	'D'
'AB'	'CDCD'	'CD'	'CDD'	'D'
'AB'	'CD'	'CD'	'CDCDD'	'D'
'AB'	''	'CD'	'CDCDCDD'	'D'

On Semantics of the Pattern Matching

n-Ary Functions

Given strings $d_1, \ldots, d_n$, matching $f(t_1, \ldots, t_n)$ against $f(d_1, \ldots, d_n)$ leads to solving the following system of equations in the free monoid:

$$\begin{cases} t_1 & = & d_1 \\ & \ldots & \\ t_n & = & d_n \end{cases}$$

Word Equations of General Form

The Language $\mathcal{L}$

$\langle \tau, R \rangle$ checks whether two given input data are equal

$\tau =$ `f('a' ++ #e.p,#e.p ++ 'a')` and R is:

```
f(e.x,e.x) = 'T';
f(e.x,e.y) = 'F';
```

The extended pattern matching has to solve the following system of the parameterized equations:

$$\begin{cases} \texttt{e.x} = \texttt{'a' ++ \#e.p} \\ \texttt{e.x} = \texttt{\#e.p ++ 'a'} \end{cases}$$

We have: `'a'++ #e.p = #e.p ++'a'`

Given two `P,Q` parameterized terms, $\tau =$ `f(P,Q)` leads to `P = Q`

Word Equations of General Form

The Language $\mathcal{L}$

$\langle \tau, R \rangle$ checks whether two given input data are equal

τ = `f('a' ++ #e.p,#e.p ++ 'a')` and R is:

```
f(e.x,e.x) = 'T';
f(e.x,e.y) = 'F';
```

The extended pattern matching has to solve the following system of the parameterized equations:

$$\begin{cases} \texttt{e.x} = \texttt{'a'} \texttt{ ++ } \texttt{\#e.p} \\ \texttt{e.x} = \texttt{\#e.p} \texttt{ ++ } \texttt{'a'} \end{cases}$$

We have: `'a'++ #e.p = #e.p ++'a'`

Given two `P,Q` parameterized terms, τ = `f(P,Q)` leads to `P = Q`

Word Equations of General Form

The Language $\mathcal{L}$

$\langle \tau, R \rangle$ checks whether two given input data are equal

τ = `f('a' ++ #e.p,#e.p ++ 'a')` and R is:

```
f(e.x,e.x) = 'T';
f(e.x,e.y) = 'F';
```

The extended pattern matching has to solve the following system of the parameterized equations:

$$\begin{cases} \texttt{e.x} = \texttt{'a' ++ \#e.p} \\ \texttt{e.x} = \texttt{\#e.p ++ 'a'} \end{cases}$$

We have: `'a'++ #e.p = #e.p ++'a'`
Given two `P,Q` parameterized terms, τ = `f(P,Q)` leads to `P = Q`

Syntactical Restriction on the Languages $\mathcal{L}, \mathcal{L}_*$

The relation `'a'++ #e.p = #e.p ++'a'` cannot be represented by using at most finitely many of the patterns to define a one-step program.

A recursion should be used to check whether an input data belongs to the truth set. This recursion is unfolded as an infinite tree.

Henceforth, we impose a restriction on $\mathcal{L}, \mathcal{L}_*$:

Syntactical Restriction on the Languages $\mathcal{L}, \mathcal{L}_*$

The relation `'a'++ #e.p = #e.p ++'a'` cannot be represented by using at most finitely many of the patterns to define a one-step program.
A recursion should be used to check whether an input data belongs to the truth set. This recursion is unfolded as an infinite tree.

Henceforth, we impose a restriction on $\mathcal{L}, \mathcal{L}_*$:

- given a program $P = \langle \pi, R \rangle$ we require for all $(\mathtt{l} = \mathtt{r}) \in R$ and for all e-variables `e.v` in $\mathtt{l}$, $\mu_{\mathtt{e.v}}(\mathtt{l}) = 1$ (the multiplicity `e.v` in $\mathtt{l}$);
- We do not impose any restriction on uses of the s-variables.

Syntactical Restriction on the Languages $\mathcal{L}, \mathcal{L}_*$

The relation `'a'++ #e.p = #e.p ++'a'` cannot be represented by using at most finitely many of the patterns to define a one-step program.
A recursion should be used to check whether an input data belongs to the truth set. This recursion is unfolded as an infinite tree.

Henceforth, we impose a restriction on $\mathcal{L}, \mathcal{L}_*$:

- given a program $P = \langle \tau, R \rangle$ we require for all `(l = r)` $\in R$ and for all e-variables `e.v` in `l`, $\mu_{e.v}(\texttt{l}) = 1$ (the multiplicity `e.v` in `l`);
- We do not impose any restriction on uses of the s-variables.

Syntactical Restriction on the Languages $\mathcal{L}, \mathcal{L}_*$

The relation `'a'++ #e.p = #e.p ++'a'` cannot be represented by using at most finitely many of the patterns to define a one-step program.
A recursion should be used to check whether an input data belongs to the truth set. This recursion is unfolded as an infinite tree.

Henceforth, we impose a restriction on $\mathcal{L}, \mathcal{L}_*$:

- given a program $P = \langle \tau, R \rangle$ we require for all `(l = r)` $\in R$ and for all e-variables `e.v` in `l`, $\mu_{e.v}(\texttt{l}) = 1$ (the multiplicity `e.v` in `l`);
- We do not impose any restriction on uses of the s-variables.

Syntactical Restriction on the Languages $\mathcal{L}, \mathcal{L}_*$

The relation `'a'++ #e.p = #e.p ++'a'` cannot be represented by using at most finitely many of the patterns to define a one-step program.
A recursion should be used to check whether an input data belongs to the truth set. This recursion is unfolded as an infinite tree.

Henceforth, we impose a restriction on $\mathcal{L}, \mathcal{L}_*$:

- given a program $P = \langle \tau, R \rangle$ we require for all `(l = r)` $\in R$ and for all e-variables `e.v` in `l`, $\mu_{e.v}(\texttt{l}) = 1$ (the multiplicity `e.v` in `l`);
- We do not impose any restriction on uses of the s-variables.

Syntactical Restriction on the Languages $\mathcal{L}, \mathcal{L}_*$

- given a program $P = \langle \tau, R \rangle$ we require for all $(\mathtt{l} = \mathtt{r}) \in R$ and for all e-variables $\mathtt{e.v}$ in $\mathtt{l}$, $\mu_{e.v}(\mathtt{l}) = 1$ (the multiplicity $\mathtt{e.v}$ in $\mathtt{l}$);
- We do not impose any restriction on uses of the s-variables.

$$\Downarrow$$

Each e-variable cannot simultaneously appear in the both sides of the corresponding word equations.

Denote the restricted versions of $\mathcal{L}, \mathcal{L}_*$ by $\mathcal{M}, \mathcal{M}_*$.

The e-Variables are Separated from the e-Parameters

by the Equality Sign / $\mathcal{M}$

Separated e-variables

τ = `f(#e.p ++ 'a' ++ #e.q)` and R is:

```
f(e.x ++ 'a' ++ e.y) = e.x ++ 'b' ++ e.y;
f(e.x) = e.x;
```

We have to solve the equation

```
e.x ++'a' ++ e.y = #e.p ++'a' ++ #e.q
```

The e-Variables are Separated from the e-Parameters

by the Equality Sign / $\mathcal{M}$

Separated e-variables

τ = `f(#e.p ++ 'a' ++ #e.q)` and R is:

```
f(e.x ++ 'a' ++ e.y) = e.x ++ 'b' ++ e.y;
f(e.x) = e.x;
```

We have to solve the equation

```
e.x ++'a' ++ e.y = #e.p ++'a' ++ #e.q
```

The e-Variables are Separated from the e-Parameters

by the Equality Sign / $\mathcal{M}$

Separated e-variables

τ = `f(#e.p ++ 'a' ++ #e.q)` and R is:

```
f(e.x ++ 'a' ++ e.y) = e.x ++ 'b' ++ e.y;
f(e.x) = e.x;
```

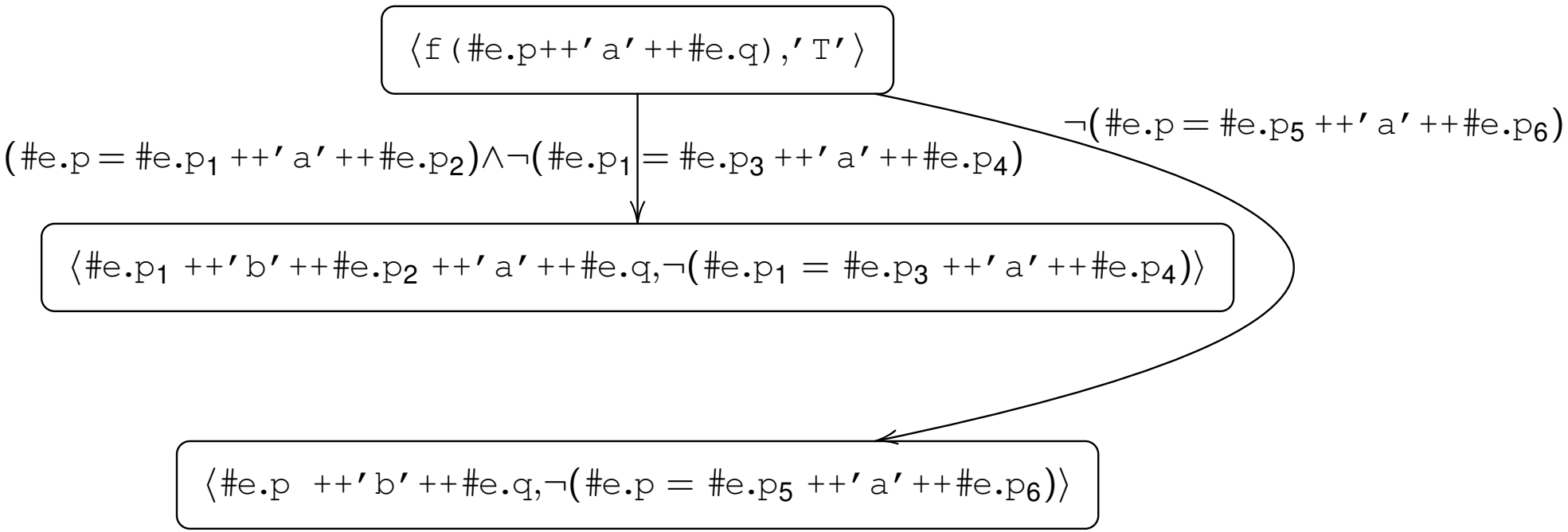
We have to solve the equation

```
e.x ++'a' ++ e.y = #e.p ++'a' ++ #e.q
```

The Result of the One-Step Unfolding

The e-Variables are Separated from the e-Parameters by the Equality Sign / $\mathcal{M}$

```
e.x ++'a' ++ e.y = #e.p ++'a' ++ #e.q
```



The Result of the One-Step Unfolding

The e-Variables are Separated from the e-Parameters by the Equality Sign / $\mathcal{M}$

The Input Program

τ = `f(#e.p ++ 'a' ++ #e.q)` and R is:

```
f(e.x ++ 'a' ++ e.y) = e.x ++ 'b' ++ e.y;
f(e.x) = #e.x;
```

The Result Program

τ_1 = f_1`(#e.p,#e.q)` and R_1 is:

```
f1(#e.p1 ++ 'a' ++ #e.p2,#e.q)
                = #e.p1 ++ 'b' ++ #e.p2 ++'a' ++ #e.q;
f1(#e.p,#e.q) = #e.p ++ 'b' ++ #e.q;
```

Our Contribution

- One-step unfolding (driving) algorithm for programs written in $\mathcal{M}_*$;
- One-step unfolding (driving) algorithm for programs written in $\mathcal{M}$;
- Description of solution sets of the word equations with separated variables.

Our Contribution

- One-step unfolding (driving) algorithm for programs written in $\mathcal{M}_*$;
- One-step unfolding (driving) algorithm for programs written in $\mathcal{M}$;
- Description of solution sets of the word equations with separated variables.

Our Contribution

- One-step unfolding (driving) algorithm for programs written in $\mathcal{M}_*$;
- One-step unfolding (driving) algorithm for programs written in $\mathcal{M}$;
- Description of solution sets of the word equations with separated variables.

Nondeterministic Case: the Language $\mathcal{M}_*$

One-Rule Programs

$$\mathbf{l} = \mathbf{r}; \xRightarrow{\texttt{Drv}} \begin{array}{l} \mathbf{l_1} = \mathbf{r_1}; \\ \quad \ldots \\ \mathbf{l_n} = \mathbf{r_n}; \end{array}$$

Nondeterministic Case: the Language $\mathcal{M}_*$

One-Rule Programs

$P = \langle \tau, R \rangle$

Example Program

τ = `f(#e.w ++ 'a' ++ #e.w)` and R is:

```
f(s.x ++ e.y ++ 'a' ++ e.z ++ s.x) = ...;
```

We have to solve the equation

```
s.x ++ e.y ++'a' ++ e.z ++ s.x = #e.w ++'a' ++ #e.w
```

Nondeterministic Case: the Language $\mathcal{M}_*$

s.x ++ e.y ++'a' ++ e.z ++ s.x = #e.w ++'a' ++ #e.w / (1)

s.x ++ ↓e.y ++'a' ++ e.z↓ ++ s.x = #e.w ++'a' ++ #e.w

Case: #e.w $\overset{\text{is}}{\to}$ λ

s.x ++ e.y ++'a' ++ e.z ++ s.x
= 'a'
s.x := 'a';
'a' ++ e.y ++'a' ++ e.z ++'a' = 'a'
e.y ++'a' ++ e.z ++'a' = λ - contr.

Case: #e.w $\overset{\text{is}}{\to}$ #s.u_1 ++ #e.w_1

s.x ++ e.y ++'a' ++ e.z ++ s.x
= #s.u_1 ++ #e.w_1 ++'a' ++ #s.u_1 ++ #e.w_1
s.x := #s.u_1;

Nondeterministic Case: the Language $\mathcal{M}_*$

s.x ++ e.y ++'a' ++ e.z ++ s.x = #e.w ++'a' ++ #e.w / (1)

s.x ++ ↓e.y ++'a' ++ e.z↓ ++ s.x = #e.w ++'a' ++ #e.w

Case: #e.w $\overset{\text{is}}{\to}$ λ

s.x ++ e.y ++'a' ++ e.z ++ s.x
= 'a'
s.x := 'a';
'a' ++ e.y ++'a' ++ e.z ++'a' = 'a'
e.y ++'a' ++ e.z ++'a' = λ - contr.

Case: #e.w $\overset{\text{is}}{\to}$ #s.u_1 ++ #e.w_1

s.x ++ e.y ++'a' ++ e.z ++ s.x
= #s.u_1 ++ #e.w_1 ++'a' ++ #s.u_1 ++ #e.w_1
s.x := #s.u_1;

Nondeterministic Case: the Language $\mathcal{M}_*$

s.x ++ e.y ++'a' ++ e.z ++ s.x = #e.w ++'a' ++ #e.w / (1)

s.x ++ ↓e.y ++'a' ++ e.z↓ ++ s.x = #e.w ++'a' ++ #e.w

Case: #e.w $\overset{\text{is}}{\to}$ λ

s.x ++ e.y ++'a' ++ e.z ++ s.x
= 'a'
s.x := 'a';
'a' ++ e.y ++'a' ++ e.z ++'a' = 'a'
e.y ++'a' ++ e.z ++'a' = λ - contr.

Case: #e.w $\overset{\text{is}}{\to}$ #s.u_1 ++ #e.w_1

s.x ++ e.y ++'a' ++ e.z ++ s.x
= #s.u_1 ++ #e.w_1 ++'a' ++ #s.u_1 ++ #e.w_1
s.x := #s.u_1;

Nondeterministic Case: the Language $\mathcal{M}_*$

s.x ++ e.y ++'a' ++ e.z ++ s.x = #e.w ++'a' ++ #e.w / (1)

s.x ++ ↓e.y ++'a' ++ e.z↓ ++ s.x = #e.w ++'a' ++ #e.w

Case: #e.w $\overset{\text{is}}{\to}$ λ

s.x ++ e.y ++'a' ++ e.z ++ s.x
= 'a'
s.x := 'a';
'a' ++ e.y ++'a' ++ e.z ++'a' = 'a'
e.y ++'a' ++ e.z ++'a' = λ - contr.

Case: #e.w $\overset{\text{is}}{\to}$ #s.u_1 ++ #e.w_1

s.x ++ e.y ++'a' ++ e.z ++ s.x
= #s.u_1 ++ #e.w_1 ++'a' ++ #s.u_1 ++ #e.w_1
s.x := #s.u_1;

Nondeterministic Case: the Language $\mathcal{M}_*$

s.x ++ e.y ++'a' ++ e.z ++ s.x = #e.w ++'a' ++ #e.w / (1)

s.x ++ ↓e.y ++'a' ++ e.z↓ ++ s.x = #e.w ++'a' ++ #e.w

Case: #e.w $\overset{\text{is}}{\to}$ λ

s.x ++ e.y ++'a' ++ e.z ++ s.x
= 'a'
s.x := 'a';
'a' ++ e.y ++'a' ++ e.z ++'a' = 'a'
e.y ++'a' ++ e.z ++'a' = λ - contr.

Case: #e.w $\overset{\text{is}}{\to}$ #s.u_1 ++ #e.w_1

s.x ++ e.y ++'a' ++ e.z ++ s.x
= #s.u_1 ++ #e.w_1 ++'a' ++ #s.u_1 ++ #e.w_1
s.x := #s.u_1;

Nondeterministic Case: the Language $\mathcal{M}_*$

s.x ++ e.y ++'a' ++ e.z ++ s.x = #e.w ++'a' ++ #e.w / (1)

s.x ++ ↓e.y ++'a' ++ e.z↓ ++ s.x = #e.w ++'a' ++ #e.w

Case: #e.w $\overset{\text{is}}{\to}$ λ

s.x ++ e.y ++'a' ++ e.z ++ s.x
= 'a'
s.x := 'a';
'a' ++ e.y ++'a' ++ e.z ++'a' = 'a'
e.y ++'a' ++ e.z ++'a' = λ - contr.

Case: #e.w $\overset{\text{is}}{\to}$ #s.u_1 ++ #e.w_1

s.x ++ e.y ++'a' ++ e.z ++ s.x
= #s.u_1 ++ #e.w_1 ++'a' ++ #s.u_1 ++ #e.w_1
s.x := #s.u_1;

Nondeterministic Case: the Language $\mathcal{M}_*$

s.x ++ e.y ++'a' ++ e.z ++ s.x = #e.w ++'a' ++ #e.w / (2)

$$\downarrow \texttt{e.y ++'a'++ e.z}\downarrow \texttt{ ++ \#s.u}_1 = \texttt{\#e.w}_1 \texttt{++'a'++ \#s.u}_1 \texttt{++ \#e.w}_1$$

Case: $\texttt{\#e.w}_1 \overset{\text{is}}{\rightarrow} \lambda$

$$\texttt{e.y ++'a' ++ e.z ++ \#s.u}_1 = \texttt{'a' ++ \#s.u}_1$$

$$\texttt{e.y ++'a' ++ e.z} = \texttt{'a'}$$

$$\texttt{e.y} := \lambda;\ \texttt{e.z} := \lambda;$$

Case: $\texttt{\#e.w}_1 \overset{\text{is}}{\rightarrow} \texttt{\#e.w}_2 \texttt{ ++ \#s.u}_2$

$$\texttt{e.y ++'a' ++ e.z ++ \#s.u}_1 = \texttt{\#e.w}_2 \texttt{++ \#s.u}_2 \texttt{++'a'++\#s.u}_1 \texttt{++ \#e.w}_2 \texttt{++\#s.u}_2$$

Nondeterministic Case: the Language $\mathcal{M}_*$

s.x ++ e.y ++'a' ++ e.z ++ s.x = #e.w ++'a' ++ #e.w / (2)

$$\downarrow \texttt{e.y ++'a'++ e.z} \downarrow \texttt{ ++ \#s.u}_1 = \texttt{ \#e.w}_1 \texttt{++'a'++ \#s.u}_1 \texttt{++ \#e.w}_1$$

Case: $\texttt{\#e.w}_1 \overset{\text{is}}{\rightarrow} \lambda$

$$\texttt{e.y ++'a' ++ e.z ++ \#s.u}_1 = \texttt{'a' ++ \#s.u}_1$$

$$\texttt{e.y ++'a' ++ e.z} = \texttt{'a'}$$

$$\texttt{e.y} := \lambda;\ \texttt{e.z} := \lambda;$$

Case: $\texttt{\#e.w}_1 \overset{\text{is}}{\rightarrow} \texttt{\#e.w}_2 \texttt{ ++ \#s.u}_2$

$$\texttt{e.y ++'a' ++ e.z ++ \#s.u}_1 = \texttt{\#e.w}_2 \texttt{++ \#s.u}_2 \texttt{++'a'++\#s.u}_1 \texttt{++ \#e.w}_2 \texttt{++\#s.u}_2$$

Nondeterministic Case: the Language $\mathcal{M}_*$

s.x ++ e.y ++'a' ++ e.z ++ s.x = #e.w ++'a' ++ #e.w / (2)

$$\downarrow \texttt{e.y ++'a'++ e.z} \downarrow \texttt{ ++ \#s.u}_1 = \texttt{ \#e.w}_1 \texttt{++'a'++ \#s.u}_1 \texttt{++ \#e.w}_1$$

Case: $\texttt{\#e.w}_1 \overset{\text{is}}{\rightarrow} \lambda$

$$\texttt{e.y ++'a' ++ e.z ++ \#s.u}_1 = \texttt{'a' ++ \#s.u}_1$$

$$\texttt{e.y ++'a' ++ e.z} = \texttt{'a'}$$

$$\texttt{e.y} := \lambda;\ \texttt{e.z} := \lambda;$$

Case: $\texttt{\#e.w}_1 \overset{\text{is}}{\rightarrow} \texttt{\#e.w}_2 \texttt{ ++ \#s.u}_2$

$$\texttt{e.y ++'a' ++ e.z ++ \#s.u}_1 = \texttt{\#e.w}_2 \texttt{++ \#s.u}_2 \texttt{++'a'++\#s.u}_1 \texttt{++ \#e.w}_2 \texttt{++\#s.u}_2$$

Nondeterministic Case: the Language $\mathcal{M}_*$

s.x ++ e.y ++'a' ++ e.z ++ s.x = #e.w ++'a' ++ #e.w / (2)

$$\downarrow \texttt{e.y ++'a'++ e.z}\downarrow \texttt{ ++ \#s.u}_1 = \texttt{\#e.w}_1\texttt{++'a'++ \#s.u}_1\texttt{++ \#e.w}_1$$

Case: $\texttt{\#e.w}_1 \overset{\text{is}}{\rightarrow} \lambda$

$$\texttt{e.y ++'a' ++ e.z ++ \#s.u}_1 = \texttt{'a' ++ \#s.u}_1$$

$$\texttt{e.y ++'a' ++ e.z} = \texttt{'a'}$$

$$\texttt{e.y} := \lambda;\ \texttt{e.z} := \lambda;$$

Case: $\texttt{\#e.w}_1 \overset{\text{is}}{\rightarrow} \texttt{\#e.w}_2\texttt{ ++ \#s.u}_2$

$$\texttt{e.y ++'a' ++ e.z ++ \#s.u}_1 = \texttt{\#e.w}_2\texttt{++ \#s.u}_2\texttt{++'a'++\#s.u}_1\texttt{++ \#e.w}_2\texttt{++\#s.u}_2$$

Nondeterministic Case: the Language $\mathcal{M}_*$

s.x ++ e.y ++'a' ++ e.z ++ s.x = #e.w ++'a' ++ #e.w / (2)

$$\downarrow \texttt{e.y ++'a'++ e.z}\downarrow \texttt{ ++ \#s.u}_1 = \texttt{\#e.w}_1 \texttt{++'a'++ \#s.u}_1 \texttt{++ \#e.w}_1$$

$$\underline{\texttt{Case:}}\ \texttt{\#e.w}_1 \overset{\texttt{is}}{\rightarrow} \lambda$$

$$\texttt{e.y ++'a' ++ e.z ++ \#s.u}_1 = \texttt{'a' ++ \#s.u}_1$$

$$\underline{\texttt{e.y := } \lambda\texttt{; e.z := } \lambda\texttt{;}}$$

$$\underline{\texttt{Case:}}\ \texttt{\#e.w}_1 \overset{\texttt{is}}{\rightarrow} \texttt{\#e.w}_2 \texttt{ ++ \#s.u}_2$$

$$\texttt{e.y ++'a' ++ e.z ++ \#s.u}_1 = \texttt{\#e.w}_2 \texttt{++ \#s.u}_2 \texttt{++'a'++\#s.u}_1 \texttt{++ \#e.w}_2 \texttt{++\#s.u}_2$$

Nondeterministic Case: the Language $\mathcal{M}_*$

s.x ++ e.y ++'a' ++ e.z ++ s.x = #e.w ++'a' ++ #e.w / (3)

$$\downarrow \texttt{e.y ++ 'a' ++ e.z} \downarrow \texttt{ ++ \#s.}u_1$$
$$= \texttt{\#e.}w_2 \texttt{ ++ \#s.}u_2 \texttt{ ++ 'a' ++ \#s.}u_1 \texttt{ ++ \#e.}w_2 \texttt{ ++ \#s.}u_2$$

Case: $\texttt{\#s.}u_2 \overset{\texttt{is}}{\rightarrow} \texttt{\#s.}u_1$

$$\texttt{e.y ++ 'a' ++ e.z ++ \#s.}u_1$$
$$= \texttt{\#e.}w_2\texttt{++\#s.}u_1\texttt{++'a'++\#s.}u_1\texttt{++ \#e.}w_2\texttt{++ \#s.}u_1$$
$$\texttt{e.y ++ 'a' ++ e.z}$$
$$= \texttt{\#e.}w_2 \texttt{ ++ \#s.}u_1 \texttt{ ++ 'a' ++ \#s.}u_1 \texttt{ ++ \#e.}w_2$$

Nondeterministic Case: the Language $\mathcal{M}_*$

s.x ++ e.y ++'a' ++ e.z ++ s.x = #e.w ++'a' ++ #e.w / (3)

$$\downarrow \texttt{e.y ++ 'a' ++ e.z} \downarrow \texttt{ ++ \#s.}u_1$$
$$= \texttt{\#e.}w_2 \texttt{ ++ \#s.}u_2 \texttt{ ++ 'a' ++ \#s.}u_1 \texttt{ ++ \#e.}w_2 \texttt{ ++ \#s.}u_2$$

Case: $\texttt{\#s.}u_2 \overset{\texttt{is}}{\rightarrow} \texttt{\#s.}u_1$

$$\texttt{e.y ++ 'a' ++ e.z ++ \#s.}u_1$$
$$= \texttt{\#e.}w_2 \texttt{++\#s.}u_1 \texttt{++'a'++\#s.}u_1 \texttt{++ \#e.}w_2 \texttt{++ \#s.}u_1$$

$$\texttt{e.y ++ 'a' ++ e.z}$$
$$= \texttt{\#e.}w_2 \texttt{ ++ \#s.}u_1 \texttt{ ++ 'a' ++ \#s.}u_1 \texttt{ ++ \#e.}w_2$$

Nondeterministic Case: the Language $\mathcal{M}_*$

s.x ++ e.y ++'a' ++ e.z ++ s.x = #e.w ++'a' ++ #e.w / (3)

$$\downarrow \texttt{e.y ++ 'a' ++ e.z} \downarrow \texttt{ ++ \#s.}u_1$$
$$= \texttt{\#e.}w_2 \texttt{ ++ \#s.}u_2 \texttt{ ++ 'a' ++ \#s.}u_1 \texttt{ ++ \#e.}w_2 \texttt{ ++ \#s.}u_2$$

Case: $\texttt{\#s.}u_2 \overset{\texttt{is}}{\rightarrow} \texttt{\#s.}u_1$

$$\texttt{e.y ++ 'a' ++ e.z ++ \#s.}u_1$$
$$= \texttt{\#e.}w_2\texttt{++\#s.}u_1\texttt{++'a'++\#s.}u_1\texttt{++ \#e.}w_2\texttt{++ \#s.}u_1$$
$$\texttt{e.y ++ 'a' ++ e.z}$$
$$= \texttt{\#e.}w_2 \texttt{ ++ \#s.}u_1 \texttt{ ++ 'a' ++ \#s.}u_1 \texttt{ ++ \#e.}w_2$$

Nondeterministic Case: the Language $\mathcal{M}_*$

s.x ++ e.y ++'a' ++ e.z ++ s.x = #e.w ++'a' ++ #e.w / (4)

$$\downarrow \texttt{e.y ++ 'a' ++ e.z} \downarrow$$
$$= \texttt{\#e.}w_2 \texttt{ ++ \#s.}u_1 \texttt{ ++ 'a' ++ \#s.}u_1 \texttt{ ++ \#e.}w_2$$

Case: $\texttt{\#e.}w_2 \overset{\text{is}}{\rightarrow} \texttt{\#e.}w_3 \texttt{ ++ 'a' ++ \#e.}w_4 \texttt{ ...}$

Case: $\texttt{\#s.}u_1 \overset{\text{is}}{\rightarrow} \texttt{'a' ...}$

Case: True

e.y := #e.w_2 ++ #s.u_1;

e.z := #s.u_1 ++ #e.w_2;

Nondeterministic Case: the Language $\mathcal{M}_*$

s.x ++ e.y ++'a' ++ e.z ++ s.x = #e.w ++'a' ++ #e.w / (4)

$$\downarrow \texttt{e.y ++ 'a' ++ e.z} \downarrow$$
$$= \#e.w_2 \texttt{ ++ } \#s.u_1 \texttt{ ++ 'a' ++ } \#s.u_1 \texttt{ ++ } \#e.w_2$$

Case: $\#e.w_2 \xrightarrow{is} \#e.w_3 \texttt{ ++ 'a' ++ } \#e.w_4 \texttt{ ...}$

Case: $\#s.u_1 \xrightarrow{is} \texttt{'a' ...}$

Case: True

e.y := #e.w2 ++ #s.u1;

e.z := #s.u1 ++ #e.w2;

Nondeterministic Case: the Language $\mathcal{M}_*$

s.x ++ e.y ++'a' ++ e.z ++ s.x = #e.w ++'a' ++ #e.w / (4)

$$\downarrow \texttt{e.y ++ 'a' ++ e.z} \downarrow$$
$$= \#e.w_2 \texttt{ ++ } \#s.u_1 \texttt{ ++ 'a' ++ } \#s.u_1 \texttt{ ++ } \#e.w_2$$

Case: $\#e.w_2 \xrightarrow{\text{is}} \#e.w_3 \texttt{ ++ 'a' ++ } \#e.w_4 \texttt{ ...}$

Case: $\#s.u_1 \xrightarrow{\text{is}} \texttt{'a' ...}$

Case: True

e.y := #e.w2 ++ #s.u1;

e.z := #s.u1 ++ #e.w2;

Nondeterministic Case: the Language $\mathcal{M}_*$

s.x ++ e.y ++'a' ++ e.z ++ s.x = #e.w ++'a' ++ #e.w / (4)

$$\downarrow \texttt{e.y ++ 'a' ++ e.z} \downarrow$$
$$= \texttt{\#e.w}_2 \texttt{ ++ \#s.u}_1 \texttt{ ++ 'a' ++ \#s.u}_1 \texttt{ ++ \#e.w}_2$$

Case: $\texttt{\#e.w}_2 \xrightarrow{\text{is}} \texttt{\#e.w}_3 \texttt{ ++ 'a' ++ \#e.w}_4 \texttt{ ...}$

Case: $\texttt{\#s.u}_1 \xrightarrow{\text{is}} \texttt{'a' ...}$

Case: True

$$\texttt{e.y := \#e.w}_2 \texttt{ ++ \#s.u}_1\texttt{;}$$
$$\texttt{e.z := \#s.u}_1 \texttt{ ++ \#e.w}_2\texttt{;}$$

Nondeterministic Case: the Language $\mathcal{M}_*$

s.x ++ e.y ++'a' ++ e.z ++ s.x = #e.w ++'a' ++ #e.w / (5)

$$\downarrow \texttt{e.y ++ 'a' ++ e.z} \downarrow$$
$$= \texttt{\#e.}w_2 \texttt{ ++ \#s.}u_1 \texttt{ ++ 'a' ++ \#s.}u_1 \texttt{ ++ \#e.}w_2$$

Case: $\texttt{\#e.}w_2 \overset{\text{is}}{\rightarrow} \texttt{\#e.}w_3 \texttt{ ++ 'a' ++ \#e.}w_4$

$$\texttt{e.y ++ 'a' ++ e.z}$$
$$= \texttt{\#e.}w_3 \texttt{ ++ 'a' ++ \#e.}w_4 \texttt{++ \#s.}u_1 \texttt{++ 'a' ++}$$
$$\texttt{++ \#s.}u_1 \texttt{ ++ \#e.}w_3 \texttt{ ++ 'a' ++ \#e.}w_4$$

Nondeterministic Case: the Language $\mathcal{M}_*$

s.x ++ e.y ++'a' ++ e.z ++ s.x = #e.w ++'a' ++ #e.w / (5)

$$\downarrow \texttt{e.y ++ 'a' ++ e.z} \downarrow$$
$$= \#e.w_2 \texttt{ ++ } \#s.u_1 \texttt{ ++ 'a' ++ } \#s.u_1 \texttt{ ++ } \#e.w_2$$

Case: $\#e.w_2 \overset{\text{is}}{\rightarrow} \#e.w_3 \texttt{ ++ 'a' ++ } \#e.w_4$

$$\texttt{e.y ++ 'a' ++ e.z}$$
$$= \#e.w_3 \texttt{ ++ 'a' ++ } \#e.w_4 \texttt{ ++ } \#s.u_1 \texttt{ ++ 'a' ++}$$
$$\texttt{++ } \#s.u_1 \texttt{ ++ } \#e.w_3 \texttt{ ++ 'a' ++ } \#e.w_4$$

Nondeterministic Case: the Language $\mathcal{M}_*$

s.x ++ e.y ++'a' ++ e.z ++ s.x = #e.w ++'a' ++ #e.w / (5)

$$\downarrow \texttt{e.y ++ 'a' ++ e.z} \downarrow$$
$$= \#e.w_2 \texttt{ ++ } \#s.u_1 \texttt{ ++ 'a' ++ } \#s.u_1 \texttt{ ++ } \#e.w_2$$

Case: $\#e.w_2 \overset{\text{is}}{\rightarrow} \#e.w_3 \texttt{ ++ 'a' ++ } \#e.w_4$

$$\texttt{e.y ++ 'a' ++ e.z}$$
$$= \#e.w_3 \texttt{ ++ 'a' ++ } \#e.w_4 \texttt{ ++ } \#s.u_1 \texttt{ ++ 'a' ++}$$
$$\texttt{++ } \#s.u_1 \texttt{ ++ } \#e.w_3 \texttt{ ++ 'a' ++ } \#e.w_4$$

Nondeterministic Case: the Language $\mathcal{M}_*$

s.x ++ e.y ++'a' ++ e.z ++ s.x = #e.w ++'a' ++ #e.w / (6)

$\downarrow$e.y ++ 'a' ++ e.z$\downarrow$ =
  #e.w_3 ++ 'a' ++ #e.w_4 ++ #s.u_1 ++ 'a' ++ #s.u_1 ++
  ++ #e.w_3 ++ 'a' ++ #e.w_4

Case: True
  e.y := #e.w_3;
  e.z := #e.w_4 ++ #s.u_1 ++ 'a' ++ #s.u_1 ++
    ++ #e.w_3 ++ 'a' ++ #e.w_4

Case: True
  e.y := #e.w_3 ++'a' ++ #e.w_4 ++ #s.u_1 ++'a' ++
    ++ #s.u_1 ++ #e.w_3
  e.z := #e.w_4;

Nondeterministic Case: the Language $\mathcal{M}_*$

s.x ++ e.y ++'a' ++ e.z ++ s.x = #e.w ++'a' ++ #e.w / (6)

$$\downarrow \texttt{e.y ++ 'a' ++ e.z} \downarrow =$$
$$\texttt{\#e.w}_3 \texttt{ ++ 'a' ++ \#e.w}_4 \texttt{ ++ \#s.u}_1 \texttt{ ++ 'a' ++ \#s.u}_1 \texttt{ ++}$$
$$\texttt{++ \#e.w}_3 \texttt{ ++ 'a' ++ \#e.w}_4$$

Case: True

$$\texttt{e.y := \#e.w}_3\texttt{;}$$
$$\texttt{e.z := \#e.w}_4 \texttt{ ++ \#s.u}_1 \texttt{ ++ 'a' ++ \#s.u}_1 \texttt{ ++ ++ \#e.w}_3 \texttt{ ++ 'a' ++ \#e.w}_4$$

Case: True

$$\texttt{e.y := \#e.w}_3 \texttt{ ++'a' ++ \#e.w}_4 \texttt{ ++ \#s.u}_1 \texttt{ ++'a' ++ ++ \#s.u}_1 \texttt{ ++ \#e.w}_3$$
$$\texttt{e.z := \#e.w}_4\texttt{;}$$

Nondeterministic Case: the Language $\mathcal{M}_*$

s.x ++ e.y ++'a' ++ e.z ++ s.x = #e.w ++'a' ++ #e.w / (6)

$\downarrow$e.y ++ 'a' ++ e.z$\downarrow$ =
  #e.w_3 ++ 'a' ++ #e.w_4 ++ #s.u_1 ++ 'a' ++ #s.u_1 ++
  ++ #e.w_3 ++ 'a' ++ #e.w_4

Case: True
  e.y := #e.w_3;
  e.z := #e.w_4 ++ #s.u_1 ++ 'a' ++ #s.u_1 ++
    ++ #e.w_3 ++ 'a' ++ #e.w_4

Case: True
  e.y := #e.w_3 ++'a' ++ #e.w_4 ++ #s.u_1 ++'a' ++
    ++ #s.u_1 ++ #e.w_3
  e.z := #e.w_4;

Nondeterministic Case: the Language $\mathcal{M}_*$

s.x ++ e.y ++'a' ++ e.z ++ s.x = #e.w ++'a' ++ #e.w / (7)

$$\downarrow \texttt{e.y ++ 'a' ++ e.z} \downarrow$$
$$= \texttt{\#e.}w_2 \texttt{ ++ \#s.}u_1 \texttt{ ++ 'a' ++ \#s.}u_1 \texttt{ ++ \#e.}w_2$$

Case: $\texttt{\#s.}u_1 \overset{\text{is}}{\rightarrow} \texttt{'a'}$

$$\texttt{e.y ++ 'a' ++ e.z}$$
$$= \texttt{\#e.}w_2 \texttt{ ++ 'a' ++ 'a' ++ 'a' ++ \#e.}w_2$$

Nondeterministic Case: the Language $\mathcal{M}_*$

s.x ++ e.y ++'a' ++ e.z ++ s.x = #e.w ++'a' ++ #e.w / (7)

$$\downarrow \texttt{e.y ++ 'a' ++ e.z} \downarrow$$
$$= \texttt{\#e.}w_2 \texttt{ ++ \#s.}u_1 \texttt{ ++ 'a' ++ \#s.}u_1 \texttt{ ++ \#e.}w_2$$

Case: $\texttt{\#s.}u_1 \overset{\text{is}}{\rightarrow} \texttt{'a'}$

$$\texttt{e.y ++ 'a' ++ e.z}$$
$$= \texttt{\#e.}w_2 \texttt{ ++ 'a' ++ 'a' ++ 'a' ++ \#e.}w_2$$

Nondeterministic Case: the Language $\mathcal{M}_*$

s.x ++ e.y ++'a' ++ e.z ++ s.x = #e.w ++'a' ++ #e.w / (7)

$$\downarrow \texttt{e.y ++ 'a' ++ e.z} \downarrow$$
$$= \texttt{\#e.}w_2 \texttt{ ++ \#s.}u_1 \texttt{ ++ 'a' ++ \#s.}u_1 \texttt{ ++ \#e.}w_2$$

$$\underline{\texttt{Case:}}\ \texttt{\#s.}u_1 \overset{\text{is}}{\rightarrow} \texttt{'a'}$$
$$\texttt{e.y ++ 'a' ++ e.z}$$
$$= \texttt{\#e.}w_2 \texttt{ ++ 'a' ++ 'a' ++ 'a' ++ \#e.}w_2$$

Nondeterministic Case: the Language $\mathcal{M}_*$

s.x ++ e.y ++'a' ++ e.z ++ s.x = #e.w ++'a' ++ #e.w / (8)

$$\downarrow \texttt{e.y ++ 'a' ++ e.z} \downarrow \texttt{= \#e.w}_2 \texttt{ ++'a' ++'a' ++'a' ++ \#e.w}_2$$

Case: True

$$\frac{\texttt{e.y := \#e.w}_2\texttt{;}}{\texttt{e.z := 'a' ++ 'a' ++ \#e.w}_2\texttt{;}}$$

Case: True

$$\frac{\texttt{e.y := \#e.w}_2\texttt{ ++ 'a' ++ 'a';}}{\texttt{e.z := \#e.w}_2\texttt{;}}$$

Nondeterministic Case: the Language $\mathcal{M}_*$

s.x ++ e.y ++'a' ++ e.z ++ s.x = #e.w ++'a' ++ #e.w / (8)

$$\downarrow \texttt{e.y ++ 'a' ++ e.z} \downarrow = \texttt{\#e.w}_2 \texttt{ ++'a' ++'a' ++'a' ++ \#e.w}_2$$

Case: True

$$\frac{\texttt{e.y := \#e.w}_2\texttt{;}}{\texttt{e.z := 'a' ++ 'a' ++ \#e.w}_2\texttt{;}}$$

Case: True

$$\frac{\texttt{e.y := \#e.w}_2\texttt{ ++ 'a' ++ 'a';}}{\texttt{e.z := \#e.w}_2\texttt{;}}$$

Nondeterministic Case: the Language $\mathcal{M}_*$

s.x ++ e.y ++'a' ++ e.z ++ s.x = #e.w ++'a' ++ #e.w / (8)

$$\downarrow \texttt{e.y ++ 'a' ++ e.z} \downarrow = \texttt{\#e.w}_2 \texttt{ ++'a' ++'a' ++'a' ++ \#e.w}_2$$

Case: True

$$\frac{\texttt{e.y := \#e.w}_2\texttt{;}}{\texttt{e.z := 'a' ++ 'a' ++ \#e.w}_2\texttt{;}}$$

Case: True

$$\frac{\texttt{e.y := \#e.w}_2\texttt{ ++ 'a' ++ 'a';}}{\texttt{e.z := \#e.w}_2\texttt{;}}$$

Nondeterministic Case: the Language $\mathcal{M}_*$

s.x ++ e.y ++'a' ++ e.z ++ s.x = #e.w ++'a' ++ #e.w / (9)

The Tree:

Case: #e.w $\overset{\text{is}}{\rightarrow}$ #s.u_1 ++ #e.w_1; s.x := #s.u_1;

Case: #e.w_1 $\overset{\text{is}}{\rightarrow}$ #e.w_2 ++ #s.u_2

Case: #s.u_2 $\overset{\text{is}}{\rightarrow}$ #s.u_1

Case: #e.w_2 $\overset{\text{is}}{\rightarrow}$ #e.w_3 ++ 'a' ++ #e.w_4

...

Case: #s.u_1 $\overset{\text{is}}{\rightarrow}$ 'a'

...

Case: True; e.y := #e.w_2 ++ #s.u_1; e.z := #s.u_1 ++ #e.w_2;

Case: #e.w_1 $\overset{\text{is}}{\rightarrow}$ λ; e.y := λ; e.z := λ;

Nondeterministic Case: the Language $\mathcal{M}_*$

s.x ++ e.y ++'a' ++ e.z ++ s.x = #e.w ++'a' ++ #e.w / (9)

The Tree:

Case: #e.w $\overset{\text{is}}{\rightarrow}$ #s.u_1 ++ #e.w_1; s.x := #s.u_1;

Case: #e.w_1 $\overset{\text{is}}{\rightarrow}$ #e.w_2 ++ #s.u_2

Case: #s.u_2 $\overset{\text{is}}{\rightarrow}$ #s.u_1

Case: #e.w_2 $\overset{\text{is}}{\rightarrow}$ #e.w_3 ++ 'a' ++ #e.w_4

...

Case: #s.u_1 $\overset{\text{is}}{\rightarrow}$ 'a'

...

Case: True; e.y := #e.w_2 ++ #s.u_1;
e.z := #s.u_1 ++ #e.w_2;

Case: #e.w_1 $\overset{\text{is}}{\rightarrow}$ λ; e.y := λ; e.z := λ;

Nondeterministic Case: the Language $\mathcal{M}_*$

s.x ++ e.y ++'a' ++ e.z ++ s.x = #e.w ++'a' ++ #e.w / (9)

The Tree:

Case: #e.w $\overset{\text{is}}{\rightarrow}$ #s.u_1 ++ #e.w_1; s.x := #s.u_1;

Case: #e.w_1 $\overset{\text{is}}{\rightarrow}$ #e.w_2 ++ #s.u_2

Case: #s.u_2 $\overset{\text{is}}{\rightarrow}$ #s.u_1

Case: #e.w_2 $\overset{\text{is}}{\rightarrow}$ #e.w_3 ++ 'a' ++ #e.w_4

...

Case: #s.u_1 $\overset{\text{is}}{\rightarrow}$ 'a'

...

Case: True; e.y := #e.w_2 ++ #s.u_1;
e.z := #s.u_1 ++ #e.w_2;

Case: #e.w_1 $\overset{\text{is}}{\rightarrow}$ λ; e.y := λ; e.z := λ;

Nondeterministic Case: the Language $\mathcal{M}_*$

s.x ++ e.y ++'a' ++ e.z ++ s.x = #e.w ++'a' ++ #e.w / (10)

The Tree (Composition):

Case: #e.w $\overset{\text{is}}{\rightarrow}$ #s.u_1 ++ #e.w_1; s.x := #s.u_1;

Case: #e.w_1 $\overset{\text{is}}{\rightarrow}$ #e.w_2 ++ #s.u_1

Case: #e.w_2 $\overset{\text{is}}{\rightarrow}$ #e.w_3 ++ 'a' ++ #e.w_4

...

Case: #s.u_1 $\overset{\text{is}}{\rightarrow}$ 'a'

Case: True; e.y := #e.w_2;
e.z := 'a' ++'a' ++ #e.w_2;

Case: True; e.y := #e.w_2 ++'a' ++ 'a';
e.z := #e.w_2;

Case: True; e.y := #e.w_2 ++ #s.u_1;
e.z := #s.u_1 ++ #e.w_2;

Case: #e.w_1 $\overset{\text{is}}{\rightarrow}$ λ; e.y := λ; e.z := λ;

Nondeterministic Case: the Language $\mathcal{M}_*$

s.x ++ e.y ++'a' ++ e.z ++ s.x = #e.w ++'a' ++ #e.w / (11)

The Tree (Composition-2):

Case: #e.w $\overset{\text{is}}{\rightarrow}$ #s.u_1 ++ #e.w_1; s.x := #s.u_1;

Case: #e.w_1 $\overset{\text{is}}{\rightarrow}$ #e.w_2 ++ #s.u_1

Case: #e.w_2 $\overset{\text{is}}{\rightarrow}$ #e.w_3 ++ 'a' ++ #e.w_4

...

Case: #s.u_1 $\overset{\text{is}}{\rightarrow}$ 'a'; e.y := #e.w_2;
e.z := 'a' ++'a' ++ #e.w_2;

Case: #s.u_1 $\overset{\text{is}}{\rightarrow}$ 'a'; e.y := #e.w_2 ++'a' ++ 'a';
e.z := #e.w_2;

Case: True; e.y := #e.w_2 ++ #s.u_1;
e.z := #s.u_1 ++ #e.w_2;

Case: #e.w_1 $\overset{\text{is}}{\rightarrow}$ λ; e.y := λ; e.z := λ;

Nondeterministic Case: the Language $\mathcal{M}_*$

s.x ++ e.y ++'a' ++ e.z ++ s.x = #e.w ++'a' ++ #e.w / (12)

Case: #e.w $\overset{\text{is}}{\rightarrow}$ #s.u_1 ++ #e.w_3 ++ 'a' ++ #e.w_4 ++ #s.u_1

s.x := #s.u_1; e.y := #e.w_3;

e.z := #e.w_4++#s.u_1++'a'++#s.u_1++#e.w_3++'a' ++ #e.w_4

Case: #e.w $\overset{\text{is}}{\rightarrow}$ #s.u_1 ++ #e.w_3 ++ 'a' ++ #e.w_4 ++ #s.u_1

s.x := #s.u_1; e.z := #e.w_4;

e.y := #e.w_3++'a'++#e.w_4++#s.u_1++'a'++ #s.u_1++ #e.w_3

Case: #e.w $\overset{\text{is}}{\rightarrow}$ 'a' ++ #e.w_2 ++ 'a';

s.x := 'a'; e.y := #e.w_2; e.z := 'aa' ++ #e.w_2;

Case: #e.w $\overset{\text{is}}{\rightarrow}$ 'a' ++ #e.w_2 ++ 'a';

s.x := 'a'; e.y := #e.w_2 ++ 'aa'; e.z := #e.w_2;

Case: #e.w $\overset{\text{is}}{\rightarrow}$ #s.u_1 ++ #e.w_2 ++ #s.u_1;

s.x := #s.u_1; e.y := #e.w_2++ #s.u_1; e.z := #s.u_1++ #e.w_2;

Case: #e.w $\overset{\text{is}}{\rightarrow}$ #s.u_1; s.x := #s.u_1; e.y := λ; e.z := λ;

Nondeterministic Case: the Language $\mathcal{M}_*$

One-Rule Programs

Input Program $P = \langle \tau, R \rangle$

$\tau =$ `f(#e.w ++ 'a' ++ #e.w)` and R is:

```
f(s.x ++ e.y ++ 'a' ++ e.z ++ s.x) = ...;
```

Residual Program $P_1 = \langle \tau_1, R_1 \rangle$

$\tau_1 =$ `f`$_1$`(#e.w)` and R_1 is:

```
f1(s.u1 ++ e.w3 ++ 'a' ++ e.w4 ++ s.u1) = ...;
f1(s.u1 ++ e.w3 ++ 'a' ++ e.w4 ++ s.u1) = ...;
f1('a' ++ e.w2 ++ 'a') = ...;
f1('a' ++ e.w2 ++ 'a') = ...;
f1(s.u1 ++ e.w2 ++ s.u1) = ...;
f1(s.u1) = ...;
```

The variables `e.y`, `e.w`$_3$ take on any string.

Nondeterministic Case: the Language $\mathcal{M}_*$

One-Rule Programs

Input Program $P = \langle \tau, R \rangle$

$\tau =$ `f(#e.w ++ 'a' ++ #e.w)` and R is:

```
f(s.x ++ e.y ++ 'a' ++ e.z ++ s.x) = ...;
```

Residual Program $P_1 = \langle \tau_1, R_1 \rangle$

$\tau_1 =$ `f`$_1$`(#e.w)` and R_1 is:

```
f1(s.u1 ++ e.w3 ++ 'a' ++ e.w4 ++ s.u1) = ...;
f1(s.u1 ++ e.w3 ++ 'a' ++ e.w4 ++ s.u1) = ...;
f1('a' ++ e.w2 ++ 'a') = ...;
f1('a' ++ e.w2 ++ 'a') = ...;
f1(s.u1 ++ e.w2 ++ s.u1) = ...;
f1(s.u1) = ...;
```

The variables `e.y`, `e.w`$_3$ take on any string.

Nondeterministic Case: the Language $\mathcal{M}_*$

One-Rule Programs

Input Program $P = \langle \tau, R \rangle$

$\tau =$ `f(#e.w ++ 'a' ++ #e.w)` and R is:

```
f(s.x ++ e.y ++ 'a' ++ e.z ++ s.x) = ...;
```

Residual Program $P_1 = \langle \tau_1, R_1 \rangle$

$\tau_1 =$ `f`$_1$`(#e.w)` and R_1 is:

```
f1(s.u1 ++ e.w3 ++ 'a' ++ e.w4 ++ s.u1) = ...;
f1(s.u1 ++ e.w3 ++ 'a' ++ e.w4 ++ s.u1) = ...;
f1('a' ++ e.w2 ++ 'a') = ...;
f1('a' ++ e.w2 ++ 'a') = ...;
f1(s.u1 ++ e.w2 ++ s.u1) = ...;
f1(s.u1) = ...;
```

The variables `e.y`, `e.w`$_3$ take on any string.

Nondeterministic Case: the Language $\mathcal{M}_*$

One-Rule Programs

Input Program $P = \langle \tau, R \rangle$

$\tau =$ `f(#e.w ++ 'a' ++ #e.w)` and R is:

```
f(s.x ++ e.y ++ 'a' ++ e.z ++ s.x) = ...;
```

Residual Program $P_1 = \langle \tau_1, R_1 \rangle$

$\tau_1 =$ `f`$_1$`(#e.w)` and R_1 is:

```
f1(s.u1 ++ e.w3 ++ 'a' ++ e.w4 ++ s.u1) = ...;
f1(s.u1 ++ e.w3 ++ 'a' ++ e.w4 ++ s.u1) = ...;
f1('a' ++ e.w2 ++ 'a') = ...;
f1('a' ++ e.w2 ++ 'a') = ...;
f1(s.u1 ++ e.w2 ++ s.u1) = ...;
f1(s.u1) = ...;
```

The variables `e.y`, `e.w`$_3$ take on any string.

Deterministic Case: the Language $\mathcal{M}$

One-Rule Programs

Input Program $P = \langle \tau, R \rangle$

$\tau =$ `f(#e.w ++ 'a' ++ #e.w)` and R is:

```
f(s.x ++ e.y ++ 'a' ++ e.z ++ s.x) = ...;
```

Residual Program $P_1 = \langle \tau_1, R_1 \rangle$

$\tau_1 =$ `f1(#e.w)` and R_1 is:

```
f1(s.u1 ++ e.w3 ++ 'a' ++ e.w4 ++ s.u1) = ...;
f1('a' ++ e.w2 ++ 'a') = ...;
f1(s.u1 ++ e.w2 ++ s.u1) = ...;
f1(s.u1) = ...;
```

The variables `e.y`, `e.w3` take on the shortest strings.

Deterministic Case: the Language $\mathcal{M}$

One-Rule Programs

Input Program $P = \langle \tau, R \rangle$

τ = `f(#e.w ++ 'a' ++ #e.w)` and R is:

```
f(s.x ++ e.y ++ 'a' ++ e.z ++ s.x) = ...;
```

Residual Program $P_1 = \langle \tau_1, R_1 \rangle$

τ_1 = `f1(#e.w)` and R_1 is:

```
f1(s.u1 ++ e.w3 ++ 'a' ++ e.w4 ++ s.u1) = ...;
f1('a' ++ e.w2 ++ 'a') = ...;
f1(s.u1 ++ e.w2 ++ s.u1) = ...;
f1(s.u1) = ...;
```

The variables `e.y`, `e.w3` take on the shortest strings.

Conclusion

- The presented algorithm vs. Turchin's one and the SCP4 supercompiler:
  - translation from Refal into its Turing-complete subset, where the pattern-matching is able produce at most one substitution;
- The problem: the actual/physical execution time taken by the step evaluation is not uniformly bounded above by the size of the input data:
  - `f(e.x ++ 'abc' ++ e.y) = ...;`
- The functional programming language REFAL.

Conclusion

- The presented algorithm vs. Turchin's one and the SCP4 supercompiler:
  - translation from Refal into its Turing-complete subset, where the pattern-matching is able produce at most one substitution;
- The problem: the actual/physical execution time taken by the step evaluation is not uniformly bounded above by the size of the input data:
  - `f(e.x ++ 'abc' ++ e.y) = ...;`
- The functional programming language REFAL.

Conclusion

- The presented algorithm vs. Turchin's one and the SCP4 supercompiler:
  - translation from Refal into its Turing-complete subset, where the pattern-matching is able produce at most one substitution;
- The problem: the actual/physical execution time taken by the step evaluation is not uniformly bounded above by the size of the input data:
  - `f(e.x ++ 'abc' ++ e.y) = ...;`
- The functional programming language REFAL.

The Problem

Associative Concatenation

Ambiguous pattern matching:

'AB'		'CD'		'D'
'AB'	'CDCDCD'	'CD'	'D'	'D'
'AB'	'CDCD'	'CD'	'CDD'	'D'
'AB'	'CD'	'CD'	'CDCDD'	'D'
'AB'	''	'CD'	'CDCDCDD'	'D'

- The time taken by the pattern-matching is not uniformly bounded above
  - by the length of the input string.

The Problem

Associative Concatenation

Ambiguous pattern matching:

'AB'		'CD'		'D'
'AB'	'CDCDCD'	'CD'	'D'	'D'
'AB'	'CDCD'	'CD'	'CDD'	'D'
'AB'	'CD'	'CD'	'CDCDD'	'D'
'AB'	''	'CD'	'CDCDCDD'	'D'

- The time taken by the pattern-matching is not uniformly bounded above
  - by the length of the input string.

The Problem

Associative Concatenation

Ambiguous pattern matching:

'AB'		'CD'		'D'
'AB'	'CDCDCD'	'CD'	'D'	'D'
'AB'	'CDCD'	'CD'	'CDD'	'D'
'AB'	'CD'	'CD'	'CDCDD'	'D'
'AB'	''	'CD'	'CDCDCDD'	'D'

- The time taken by the pattern-matching is not uniformly bounded above
  - by the length of the input string.

Thank You