From Monadic Second-Order Definable String Transformations to Transducers

Rajeev Alur\textsuperscript{1} \hspace{1cm} Antoine Durand-Gasselin\textsuperscript{2} \hspace{1cm} Ashutosh Trivedi\textsuperscript{3}

\textsuperscript{1}University of Pennsylvania

\textsuperscript{2}LIAFA, Université Paris Diderot

\textsuperscript{3}Indian Institute of Technology Bombay

June 21, 2013
Regular Word Analysis

Qualitative properties over words

$$\varphi : \Sigma^\infty \rightarrow \{0, 1\}$$

Logically
MSO formulas

Computational model
Finite state automata
Regular Word Analysis

Qualitative properties over words
\[ \varphi : \Sigma^\infty \rightarrow \{0, 1\} \]

Logically
MSO formulas

Computational model
Finite state automata

Equivalence [Büchi, 1960]
Regular Word Analysis

Qualitative properties over words
\[ \varphi : \Sigma^\infty \rightarrow \{0, 1\} \]

Logically \hspace{1cm} Computational model

MSO formulas \hspace{1cm} Finite state automata

Equivalence [Büchi, 1960]

Regular Word Transformation

Transformations of words
\[ \varphi : \Sigma^\infty \rightarrow \Sigma^\infty \]

Logically \hspace{1cm} Computational model

Definition using MSO \hspace{1cm} Streaming Transducers
Regular Word Analysis

Qualitative properties over words
\[ \varphi : \Sigma^\infty \rightarrow \{0, 1\} \]

Logically

MSO formulas

Computational model

Finite state automata

Equivalence [Büchi, 1960]

Regular Word Transformation

Transformations of words
\[ \varphi : \Sigma^\infty \rightarrow \Sigma^\infty \]

Logically

Definition using MSO

Computational model

Streaming Transducers

Equi-expressiveness
Monadic Second Order Logic (MSO)

We deal about MSO over the linear order:

- The structure is $(N, >, P_1, \ldots, P_k)$
- The domain: $\mathbb{N}$ or $[1, n]$
- The order relation
- Some unary predicates

An MSO formula with no free variables defines a language
Monadic Second Order Logic (MSO)

We deal about MSO over the linear order:

- The structure is
  \(( \mathbb{N}, >, P_1, \ldots, P_k )\)

- The domain: \(\mathbb{N}\) or \([1, n]\)

- The order relation

- Some unary predicates

Words are interpreted structures: e.g. \(([1, 10], >, P_a, P_b, P_c)\)

- \(w = a \ b \ b \ a \ b \ c \ a \ b \ c \ c\)
- \(P_a = \{1, 4, 7\}\)
- \(P_b = \{2, 3, 5, 8\}\)
- \(P_c = \{6, 9, 10\}\)
Monadic Second Order Logic (MSO)

We deal about MSO over the linear order:

- The structure is
  \[(\mathbb{N}, >, P_1, \ldots, P_k)\]

  - The domain: \(\mathbb{N}\) or \([1, n]\)
  - The order relation
  - Some unary predicates

  Words are interpreted structures: e.g. \(([1, 10], >, P_a, P_b, P_c)\)

  \[w = a \ b \ b \ a \ b \ c \ a \ b \ c \ c\]

  \[P_a = \{1, 4, 7\}\]

  \[P_b = \{2, 3, 5, 8\}\]

  \[P_c = \{6, 9, 10\}\]

- Formulas are defined inductively:
  - Atomic: \(x_1 < x_2, P(x_1), X(x), \ldots\)
  - Boolean connectives: \(\varphi_1 \land \varphi_2, \neg \varphi_3, \ldots\)
  - First-order quantification: \(\exists x. \varphi\)
  - Second-order quantification: \(\exists X. \varphi\)
Monadic Second Order Logic (MSO)

We deal about MSO over the linear order:

- The structure is
  \[(\mathbb{N}, >, P_1, \ldots, P_k)\]

- The domain: \(\mathbb{N}\) or \([1, n]\)
- The order relation
- Some unary predicates

Words are interpreted structures: e.g. \(([1, 10], >, P_a, P_b, P_c)\)

\[w = a \ b \ b \ a \ b \ c \ a \ b \ c \ c\]

- \(P_a = \{1, 4, 7\}\)
- \(P_b = \{2, 3, 5, 8\}\)
- \(P_c = \{6, 9, 10\}\)

Formulas are defined inductively:

- Atomic: \(x_1 < x_2, P(x_1), X(x), \ldots\)
- Boolean connectives: \(\varphi_1 \land \varphi_2, \neg \varphi_3, \ldots\)
- First-order quantification: \(\exists x.\varphi\)
- Second-order quantification: \(\exists X.\varphi\)

An MSO formula with no free variables defines a language
Theorem [Büchi, 1960]

A language is MSO definable iff it is accepted by a finite-state automaton.

- Deterministic automata are a computational model to analyse words: process sequentially a word input by jumping from state to state
- Can be efficiently manipulated
  - Automata can be determinized
  - LSPACE algorithm to check if a word is accepted by an automaton
  - Minimization (equivalence in time $O(n \log \log n)$)
  - Product of automata (language union, intersection,...)
Regular Word Analysis

Qualitative properties over words
\[ \varphi : \Sigma^\infty \rightarrow \{0, 1\} \]

Logically
Computational model

MSO formulas
Finite state automata

Equivalence [Büchi, 1960]

Regular Word Transformation

Transformations of words
\[ \varphi : \Sigma^\infty \rightarrow \Sigma^\infty \]

Logically
Computational model

Definition using MSO
Streaming Transducers

Equi-expressiveness
Contents

1 Regular Transformations
   • Logical definition
   • Streaming Transducers

2 Contribution: Equivalence with a direct logic-based reduction
   • Some logical considerations
   • Proof Walkthrough

3 Decision procedures
   • Functional equivalence
   • Typechecking Problem

4 Conclusion
Contents

1 Regular Transformations
   - Logical definition
   - Streaming Transducers

2 Contribution: Equivalence with a direct logic-based reduction
   - Some logical considerations
   - Proof Walkthrough

3 Decision procedures
   - Functional equivalence
   - Typechecking Problem

4 Conclusion
MSO-definable Transformations

Courcelle, 1994] Defining Graph Transformations using MSO

A labeled graph transformation using MSO is specified by:

- **input** and **output** alphabets;
- an MSO formula specifying the **domain** of the transformation;
- output is specified using a **finite number of copies** of nodes of input graph;
- the **node labels** are specified using MSO formulas; and
- the **existence of edges** between nodes of various copies is specified using MSO formulas.
MSO-definable Transformations

[Courcelle, 1994] Defining Graph Transformations using MSO

A labeled graph transformation using MSO is specified by:

- input and output alphabets;
- an MSO formula specifying the domain of the transformation;
- output is specified using a finite number of copies of nodes of input graph;
- the node labels are specified using MSO formulas; and
- the existence of edges between nodes of various copies is specified using MSO formulas

Example

Let $\Sigma = \{a, b, \#\}$. Consider a transformation $f_1 : \Sigma^\infty \rightarrow \Sigma^\infty$

$$u_1\#u_2\# \ldots u_{n-1}\#u_n\#v \mapsto \overline{u_1}u_1\# \ldots \#\overline{u_n}u_n\#v.$$ 

where $\overline{u}$ is reverse of $u$. 
MSO-definable Transformations

input: \[ a \rightarrow b \rightarrow c \rightarrow b \rightarrow b \rightarrow \# \rightarrow a \rightarrow a \rightarrow b \rightarrow \# \rightarrow c \rightarrow c \rightarrow a \rightarrow a \rightarrow \# \rightarrow b \]

copy 1: \[ a \leftarrow b \leftarrow c \leftarrow b \leftarrow b \leftarrow a \leftarrow a \leftarrow b \leftarrow c \leftarrow c \leftarrow a \leftarrow a \leftarrow a \leftarrow b \]

copy 2: \[ a \rightarrow b \rightarrow c \rightarrow b \rightarrow b \rightarrow \# \rightarrow a \rightarrow a \rightarrow b \rightarrow \# \rightarrow c \rightarrow c \rightarrow a \rightarrow a \rightarrow \# \rightarrow b \]

\[ \Sigma = \Gamma = \{a, b, c, \#\}, \ C = \{1, 2\}, \ \text{and} \]

\[ \]
MSO-definable Transformations

input: $a \rightarrow b \rightarrow c \rightarrow b \rightarrow b \rightarrow \# \rightarrow a \rightarrow a \rightarrow b \rightarrow \# \rightarrow c \rightarrow c \rightarrow a \rightarrow a \rightarrow \# \rightarrow b$

copy 1: $a \leftarrow b \leftarrow c \leftarrow b \leftarrow b \leftarrow a \leftarrow a \leftarrow b \leftarrow c \leftarrow c \leftarrow a \leftarrow a$

copy 2: $a \rightarrow b \rightarrow c \rightarrow b \rightarrow b \rightarrow \# \rightarrow a \rightarrow a \rightarrow b \rightarrow \# \rightarrow c \rightarrow c \rightarrow a \rightarrow a \rightarrow \# \rightarrow b$

- $\Sigma = \Gamma = \{a, b, c, \#\}$, $C = \{1, 2\}$, and
- Node Label Formulas ($|\Gamma| \cdot |C|$ formulas)
  - $\text{Label}^c_1(x) = \text{Label}^{\text{inp}}_\alpha(x) \land \neg \text{Label}^{\text{inp}}_\#(x) \land \text{reach}_\#(x)$
  - $\text{Label}^c_2(x) = \text{Label}^{\text{inp}}_\alpha(x)$
MSO-definable Transformations

input:  \[ a \rightarrow b \rightarrow c \rightarrow b \rightarrow b \rightarrow \# \rightarrow a \rightarrow a \rightarrow b \rightarrow \# \rightarrow c \rightarrow c \rightarrow a \rightarrow a \rightarrow \# \rightarrow b \]

copy 1:  \[ a \leftarrow b \leftarrow c \leftarrow b \leftarrow b \leftarrow a \leftarrow a \leftarrow b \leftarrow c \leftarrow c \leftarrow a \leftarrow a \leftarrow \# \]

copy 2:  \[ a \rightarrow b \rightarrow c \rightarrow b \rightarrow b \rightarrow \# \rightarrow a \rightarrow a \rightarrow b \rightarrow \# \rightarrow c \rightarrow c \rightarrow a \rightarrow a \rightarrow \# \rightarrow b \]

- \[ \Sigma = \Gamma = \{ a, b, c, \# \} \], \[ C = \{ 1, 2 \} \], and
- Node Label Formulas (\(|\Gamma| \cdot |C|\) formulas)
  - \[ \text{Label}^{c_1}_{\alpha}(x) = \text{Label}^{\text{inp}}_{\alpha}(x) \land \neg \text{Label}^{\#}_{\text{inp}}(x) \land \text{reach}_{\#}(x) \]
  - \[ \text{Label}^{c_2}_{\alpha}(x) = \text{Label}^{\text{inp}}_{\alpha}(x) \]
- Edge Label Formulas (\(|C|^2\) formulas)
  - \[ \text{Edge}^{c_1,c_1}_{\alpha}(x, y) = \text{Edge}^{\text{inp}}_{\alpha}(y, x) \land \neg \text{Label}^{\#}_{\text{inp}}(x) \land \neg \text{Label}^{\#}_{\text{inp}}(y) \]
  - \[ \text{Edge}^{c_2,c_2}_{\alpha}(x, y) = \text{Edge}^{\text{inp}}_{\alpha}(x, y) \land (\neg \text{Label}^{\#}_{\text{inp}}(x) \lor (\text{Label}^{\#}_{\text{inp}}(x) \land \neg \text{reach}_{\#}(x))) \]
  - \[ \text{Edge}^{1,2}_{\alpha}(x, y) = (x = y) \land (\text{first}(x) \lor \exists z(\text{Label}^{\#}_{\text{inp}}(z) \land \text{Edge}^{\text{inp}}_{\alpha}(z, x))) \]
  - \[ \text{Edge}^{2,1}_{\alpha}(x, y) = \text{Label}^{\#}_{\text{inp}}(x) \land \text{reach}_{\#}(x) \land (\exists z(\text{Edge}^{\text{inp}}_{\alpha}(y, z) \land \text{Label}^{\#}_{\text{inp}}(z))) \land (\forall z((\text{path}(x, z) \land \text{path}(z, y)) \rightarrow \neg \text{Label}^{\#}_{\text{inp}}(z))) \]
Streaming Transducers [Alur and Černý, 2011]

A streaming transducer is an automaton:

\[(\Sigma, Q, \delta, q_0, \mathcal{F}, X, (D, f_1, \ldots, f_k), \rho)\]
A streaming transducer is an automaton:

\[
(\Sigma, Q, \delta, q_0, F, X, (D, f_1, \ldots, f_k), \rho)
\]

- extended with a finite set of registers \(X\)
A streaming transducer is an automaton:

$$(\Sigma, Q, \delta, q_0, F, X, (D, f_1, \ldots, f_k), \rho)$$

- extended with a finite set of registers $X$
- which will store values from domain $D$
A streaming transducer is an automaton:
\[(\Sigma, Q, \delta, q_0, \mathcal{F}, X, (D, f_1, \ldots, f_k), \rho)\]

- extended with a finite set of registers \(X\)
- which will store values from domain \(D\)
- with an update function \(\rho : Q \times \Sigma \rightarrow X \rightarrow T(X, f_1, \ldots, f_k)\)
  \(T(X, f_1, \ldots, f_k)\) denotes terms obtained with functions \(f_1, \ldots, f_k\) and registers.
A streaming transducer is an automaton:
\[(\Sigma, Q, \delta, q_0, \mathcal{F}, X, (D, f_1, \ldots, f_k), \rho)\]

- extended with a finite set of registers \(X\)
- which will store values from domain \(D\)
- with an update function \(\rho : Q \times \Sigma \rightarrow X \rightarrow \mathcal{T}(X, f_1, \ldots, f_k)\)
  \(\mathcal{T}(X, f_1, \ldots, f_k)\) denotes terms obtained with functions \(f_1, \ldots, f_k\) and registers.
- Its configurations will be a state together with a valuation of each register \(Q \times [X \rightarrow D]\)
Streaming Transducers [Alur and Černý, 2011]

A streaming transducer is an automaton:

\[(\Sigma, Q, \delta, q_0, \mathcal{F}, X, (D, f_1, \ldots, f_k), \rho)\]

- extended with a finite set of registers \(X\)
- which will store values from domain \(D\)
- with an update function \(\rho: Q \times \Sigma \rightarrow X \rightarrow \mathcal{T}(X, f_1, \ldots, f_k)\)
  \(\mathcal{T}(X, f_1, \ldots, f_k)\) denotes terms obtained with functions \(f_1, \ldots, f_k\) and registers.
- Its configurations will be a state together with a valuation of each register \(Q \times [X \rightarrow D]\)
- An output function \(\mathcal{F}:\)
  - Finite word input this is a function from \(Q\) to \(\mathcal{T}(X, f_1, \ldots, f_k)\).
    The image of \(w\) is the value of the term \(\mathcal{F}(\hat{\delta}(w))\)
  - in the case of infinite word input, this is a function from \(2^Q\) to \(X\).
    The image of \(w\) is the limit of the value of register \(\mathcal{F}(\hat{\delta}(w))\)
An example

\[ A = ( \Sigma, Q, \delta, q_0, \mathcal{F}, X, (D, f_1, \ldots, f_k), \rho ) \]

Here \( D = \Sigma^* \), and we have binary function \( \cdot \) (concatenation) and constants \( \varepsilon, a, b, c, \# \)
An example

\[ A = (\Sigma, Q, \delta, q_0, \mathcal{F}, X, (D, f_1, \ldots, f_k), \rho) \]

Here \( D = \Sigma^* \), and we have binary function \( \cdot \) (concatenation) and constants \( \varepsilon, a, b, c, \# \)

\[
\begin{align*}
\text{start} & \quad \rightarrow \quad q \\
X := X \cdot Z \cdot \# & \quad \text{\#} \\
Y := \varepsilon & \\
Z := \varepsilon \\
X := X & \\
Y := Y \cdot \alpha & \\
Z := \alpha \cdot Z \cdot \alpha \\
F(q) &= X \cdot Y
\end{align*}
\]

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>#</th>
<th>a</th>
<th>#</th>
<th>a</th>
<th>b</th>
</tr>
</thead>
</table>
\(X\) | \(Y\) | \(Z\)
An example

\[ A = (\Sigma, Q, \delta, q_0, \mathcal{F}, X, (D, f_1, \ldots, f_k), \rho) \]

Here \( D = \Sigma^* \), and we have binary function \( \cdot \) (concatenation) and constants \( \varepsilon, a, b, c, \# \)

\[
\begin{align*}
\text{start} & \rightarrow q \\
X & := X \cdot Z \cdot \# \\
Y & := \varepsilon \\
Z & := \varepsilon \\
F(q) & = X \cdot Y
\end{align*}
\]

\[
\begin{array}{cccccc}
a & b & \# & a & \# & a & b \\
\hline
X & \varepsilon \\
Y & a \\
Z & aa
\end{array}
\]
An example

\[ A = \left( \Sigma, Q, \delta, q_0, \mathcal{F}, X, (D, f_1, \ldots, f_k), \rho \right) \]

Here \( D = \Sigma^* \), and we have binary function \( \cdot \) (concatenation) and constants \( \varepsilon, a, b, c, \# \)

\[
\begin{array}{c|c}
\text{start} & q \\
\hline
\# & X := X \cdot Z \cdot \# \\
\varepsilon & Y := \varepsilon \\
\varepsilon & Z := \varepsilon \\
\varepsilon & X := X \\
\alpha & Y := Y \cdot \alpha \\
\alpha & Z := \alpha \cdot Z \cdot \alpha \\
\end{array}
\]

\[ \mathcal{F}(q) = X \cdot Y \]

\[
\begin{array}{cccc}
a & b & \# & a & \# & a & b \\
X & \varepsilon & \varepsilon & \varepsilon \\
Y & a & ab \\
Z & aa & baab \\
\end{array}
\]
An example

\[ A = (\Sigma, Q, \delta, q_0, \mathcal{F}, X, (D, f_1, \ldots, f_k), \rho) \]

Here \( D = \Sigma^* \), and we have binary function \( \cdot \) (concatenation) and constants \( \varepsilon, a, b, c, \# \)

\[
\begin{align*}
\text{start} & \rightarrow q \\
X & := X \cdot Z \cdot \# \\
Y & := \varepsilon \\
Z & := \varepsilon \\
\mathcal{F}(q) &= X \cdot Y \\
X & := X \\
Y & := Y \cdot \alpha \\
Z & := \alpha \cdot Z \cdot \alpha
\end{align*}
\]

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>#</th>
<th>a</th>
<th>b</th>
<th>#</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>\varepsilon</td>
<td>\varepsilon</td>
<td>\text{baab}</td>
<td>#</td>
<td></td>
</tr>
<tr>
<td>Y</td>
<td>a</td>
<td>ab</td>
<td>\varepsilon</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Z</td>
<td>aa</td>
<td>\text{baab}</td>
<td>\varepsilon</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
An example

\[ A = (\Sigma, Q, \delta, q_0, \mathcal{F}, X, (D, f_1, \ldots, f_k), \rho) \]

Here \( D = \Sigma^* \), and we have binary function \( \cdot \) (concatenation) and constants \( \varepsilon, a, b, c, \# \)

\[ \begin{align*}
\text{start} &\rightarrow q \\
X &:= X \cdot Z \cdot \# \\
# &:= \varepsilon \\
Y &:= \varepsilon \\
Z &:= \varepsilon \\
\alpha &:= X \\
Y &:= Y \cdot \alpha \\
Z &:= \alpha \cdot Z \cdot \alpha \\
\mathcal{F}(q) &= X \cdot Y
\end{align*} \]

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>#</th>
<th>a</th>
<th>#</th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>\varepsilon</td>
<td>\varepsilon</td>
<td>baab#</td>
<td>baab#</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Y</td>
<td>a</td>
<td>ab</td>
<td>\varepsilon</td>
<td>a</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Z</td>
<td>aa</td>
<td>baab</td>
<td>\varepsilon</td>
<td>aa</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
An example

\[ A = (\Sigma, Q, \delta, q_0, \mathcal{F}, X, (D, f_1, \ldots, f_k), \rho) \]

Here \( D = \Sigma^* \), and we have binary function \( \cdot \) (concatenation) and constants \( \varepsilon, a, b, c, \# \)

\[
\begin{align*}
\text{start} & \quad \longrightarrow \quad q \\
\end{align*}
\]

\[
\begin{array}{c|c}
\text{X} & \text{X} := X \cdot Z \cdot \# \\
\text{Y} & \text{Y} := \varepsilon \\
\text{Z} & \text{Z} := \varepsilon \\
\alpha & \\
\end{array}
\]

\[ \mathcal{F}(q) = X \cdot Y \]

\[
\begin{array}{cccccc}
a & b & \# & a & \# & a & b \\
X & \varepsilon & \varepsilon & \text{baab}\# & \text{baab}\# & \text{baab}\#\varepsilon \\
Y & a & ab & \varepsilon & a & \varepsilon \\
Z & aa & \text{baab} & \varepsilon & aa & \varepsilon \\
\end{array}
\]
An example

\[ A = ( \Sigma, Q, \delta, q_0, \mathcal{F}, X, (D, f_1, \ldots, f_k), \rho ) \]

Here \( D = \Sigma^* \), and we have binary function \( \cdot \) (concatenation) and constants \( \varepsilon, a, b, c, \# \)

\[
\begin{align*}
X &:= X \cdot Z \cdot \# \\
\# &:= \varepsilon \\
Y &:= \varepsilon \\
Z &:= \varepsilon \\
X &:= X \\
\alpha &:= Y \cdot \alpha \\
Y &:= Y \cdot \alpha \\
Z &:= \alpha \cdot Z \cdot \alpha
\end{align*}
\]

\[ \mathcal{F}(q) = X \cdot Y \]

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>#</th>
<th>a</th>
<th>#</th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>ε</td>
<td>ε</td>
<td>baab#</td>
<td>baab#</td>
<td>baab#aa#</td>
<td>baab#aa#</td>
<td></td>
</tr>
<tr>
<td>Y</td>
<td>a</td>
<td>ab</td>
<td>ε</td>
<td>a</td>
<td>ε</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>Z</td>
<td>aa</td>
<td>baab</td>
<td>ε</td>
<td>aa</td>
<td>ε</td>
<td>aa</td>
<td></td>
</tr>
</tbody>
</table>
An example

\[ A = (\Sigma, Q, \delta, q_0, \mathcal{F}, X, (D, f_1, \ldots, f_k), \rho) \]

Here \( D = \Sigma^* \), and we have binary function \( \cdot \) (concatenation) and constants \( \varepsilon, a, b, c, \# \)

\[
\begin{align*}
\text{start} & \quad \rightarrow \quad q \\
X &= X \cdot Z \cdot \# \\
Y &= \varepsilon \\
Z &= \varepsilon \\
F(q) &= X \cdot Y
\end{align*}
\]

\[
\begin{array}{cccccccc}
a & b & \# & a & \# & a & b \\
X & \varepsilon & \varepsilon & \text{baab}\# & \text{baab}\# & \text{baab}\#\text{aa}\# & \text{baab}\#\text{aa}\# & \text{baab}\#\text{aa}\# \\
Y & a & ab & \varepsilon & a & \varepsilon & a & ab \\
Z & aa & \text{baab} & \varepsilon & aa & \varepsilon & aa & \text{baab}
\end{array}
\]
A restriction on the update function

\[ \alpha | X = X \cdot \alpha \cdot X \quad \mathcal{F} = X \]

This leads to an exponential output
A restriction on the update function

\[ \alpha | X = X \cdot \alpha \cdot X \quad F = X \]

- This leads to an exponential output
- We want to forbid this behaviour:
  - Copylessness: each register appear at most once on the r.h.s.
  - Restricted copy: copies allowed but recombining is not possible
  - Bounded copy: in the end the content of any register is never copies more than a bounded number of times
A restriction on the update function

This leads to an exponential output

We want to forbid this behaviour:

- Copylessness: each register appear at most once on the r.h.s.
- Restricted copy: copies allowed but recombining is not possible
- Bounded copy: in the end the content of any register is never copies more than a bounded number of times

Streaming transducers have to satisfy this syntactic restriction
Contents

1. Regular Transformations
   - Logical definition
   - Streaming Transducers

2. Contribution: Equivalence with a direct logic-based reduction
   - Some logical considerations
   - Proof Walkthrough

3. Decision procedures
   - Functional equivalence
   - Typechecking Problem

4. Conclusion
A computational model
for some restricted Courcelle transformations

\[ \begin{array}{ccccccc}
\text{from} & \{0, 1\} & \Sigma^* & \Sigma^\omega & \mathcal{T}^* & \mathcal{T}^\omega & \text{graphs} \\
\text{to} \\
\text{finite words} \\
\text{infinite words} \\
\text{finite trees} \\
\text{infinite trees} \\
\text{graphs}
\end{array} \]
A computational model
for some restricted Courcelle transformations

\[
\begin{array}{ccccccc}
\text{from} & \{0, 1\} & \Sigma^* & \Sigma^\omega & \mathcal{T}^* & \mathcal{T}^\omega & \text{graphs} \\
\text{finite words} & \text{Büchi} \\
\text{infinite words} & \text{Rabin} \\
\text{finite trees} & \text{Rabin} \\
\text{infinite trees} & \text{Rabin} \\
\text{graphs} & \text{Rabin} \\
\end{array}
\]
A computational model
for some restricted Courcelle transformations

from finite words to \{0, 1\}, \Sigma^*, \Sigma^\omega, \mathcal{T}^*, \mathcal{T}^\omega, graphs

- finite words
- Büchi [AČ11]

- infinite words

- finite trees
- Rabin

- infinite trees

- graphs

[Alur and Černý, 2011] (POPL) Streaming transducers for algorithmic verification of single-pass list-processing programs
Existing proof, through a two way transducer
Case of transformations from finite strings to finite strings [Alur and Černý, 2011]

Alur, Durand-Gasselin, Trivedi

Streaming String Transducers

June 21, 2013 15 / 31
A computational model
for some restricted Courcelle transformations

\[
\text{from } \{0, 1\} \rightarrow \Sigma^* \rightarrow \Sigma^\omega \rightarrow \mathcal{T}^* \rightarrow \mathcal{T}^\omega \rightarrow \text{graphs}
\]

finite words \quad Büchi \quad [AČ11]

infinite words \quad [AFT12]

finite trees \quad Rabin

infinite trees

graphs

- [Alur and Černý, 2011] (POPL) Streaming transducers for algorithmic verification of single-pass list-processing programs
- [Alur et al., 2012] (LICS) Regular Transformations of Infinite Strings
Existing proof, of a through way transducer
Case of transformations from infinite strings to infinite strings [Alur et al., 2012]

[Engelfriet and Hoogeboom, 2001] [Alur et al., 2012]

MSO Transformation

Two-Way transducer w/ look-ahead

Functional NSST w/ look-ahead

[Alur and Černý, 2011]

Streaming Transducer

Streaming Transducer w/ bounded copy

Functional NSST

[Miyano and Hayashi, 1984]

[Alur et al., 2012]

[Alur et al., 2012]
A computational model for some restricted Courcelle transformations

<table>
<thead>
<tr>
<th>from</th>
<th>to</th>
<th>(\Sigma^*)</th>
<th>(\Sigma^\omega)</th>
<th>(\mathcal{T}^*)</th>
<th>(\mathcal{T}^\omega)</th>
<th>graphs</th>
</tr>
</thead>
<tbody>
<tr>
<td>finite words</td>
<td>({0, 1})</td>
<td>Büchi</td>
<td>[AČ11]</td>
<td>[Ad’A12]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>infinite words</td>
<td></td>
<td></td>
<td>[AFT12]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>finite trees</td>
<td></td>
<td></td>
<td>[Ad’A12]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>infinite trees</td>
<td></td>
<td></td>
<td>[Ad’A12]</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- [Alur and Černý, 2011] (POPL) Streaming transducers for algorithmic verification of single-pass list-processing programs
- [Alur et al., 2012] (LICS) Regular Transformations of Infinite Strings
- [Alur and D’Antoni, 2012] (ICALP) Streaming Tree Transducers
A computational model
for some restricted Courcelle transformations

<table>
<thead>
<tr>
<th>from</th>
<th>to</th>
<th>{0, 1}</th>
<th>(\Sigma^*)</th>
<th>(\Sigma^\omega)</th>
<th>(\mathcal{T}^*)</th>
<th>(\mathcal{T}^\omega)</th>
<th>graphs</th>
</tr>
</thead>
<tbody>
<tr>
<td>finite words</td>
<td>Büchi</td>
<td>[AČ11]</td>
<td></td>
<td></td>
<td>[Ad’A12]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>infinite words</td>
<td></td>
<td>[AFT12]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>finite trees</td>
<td>Rabin</td>
<td>[Ad’A12]</td>
<td></td>
<td></td>
<td>[Ad’A12]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>infinite trees</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- [Alur and Černý, 2011] (POPL) Streaming transducers for algorithmic verification of single-pass list-processing programs
- [Alur et al., 2012] (LICS) Regular Transformations of Infinite Strings
- [Alur and D’Antoni, 2012] (ICALP) Streaming Tree Transducers
A computational model
for some restricted Courcelle transformations

<table>
<thead>
<tr>
<th>from</th>
<th>to</th>
<th>{0, 1}</th>
<th>(\Sigma^*)</th>
<th>(\Sigma^\omega)</th>
<th>(\mathcal{T}^*)</th>
<th>(\mathcal{T}^\omega)</th>
<th>graphs</th>
</tr>
</thead>
<tbody>
<tr>
<td>finite words</td>
<td>Büchi</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>infinite words</td>
<td></td>
<td>[AČ11]</td>
<td></td>
<td></td>
<td></td>
<td>[Ad’A12]</td>
<td></td>
</tr>
<tr>
<td>finite trees</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>[AFT12]</td>
<td>[ADT13]</td>
<td></td>
</tr>
<tr>
<td>infinite trees</td>
<td></td>
<td>[Ad’A12]</td>
<td></td>
<td></td>
<td>[Ad’A12]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>graphs</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>[Ad’A12]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- [Alur and Černý, 2011] (POPL) Streaming transducers for algorithmic verification of single-pass list-processing programs
- [Alur et al., 2012] (LICS) Regular Transformations of Infinite Strings
- [Alur and D’Antoni, 2012] (ICALP) Streaming Tree Transducers
- [Alur et al., 2013] (LICS) From Monadic Second-Order Definable String Transformations to Transducers
A direct proof

MSO Transformation

[Alur, DG and Trivedi 13]

[Alur and Černý, 2011]
[Alur et al., 2012]

Streaming Transducer w/ restricted copy
Finiteness of MSO formula up to some quantifier depth

Remark

- The number of MSO sentences of quantifier depth at most $k$ is finite.
Finiteness of MSO formula up to some quantifier depth

Remark

- The number of MSO sentences of quantifier depth at most $k$ is finite.
  
  By induction over formulas with $r$ FV and quantifier depth at most $k$:
  
  - true when $k = 0$
  - if true for some $k$, notice that an MSO formula with $qd k + 1$ and $r$ FV is a boolean combination of formulas of the form $\exists X.\varphi$ where $\varphi$ has $qd k$ and $r + 1$ FV.
  - Thus a finitely generated (by induction) boolean algebra
Finiteness of MSO formula up to some quantifier depth

Remark

- The number of MSO sentences of quantifier depth at most $k$ is finite
- We define an equivalence relation over words
  - Two words are $k$-equivalent iff no formula of q.d. $k$ can distinguish them
  - This equivalence relation has finite index
  - We denote $k$-types these equivalence classes
Finiteness of MSO formula up to some quantifier depth

Remark

- The number of MSO sentences of quantifier depth at most $k$ is finite
- We define an equivalence relation over words
  - Two words are $k$-equivalent iff no formula of q.d. $k$ can distinguish them
  - This equivalence relation has finite index
  - We denote $k$-types these equivalence classes

Remark: This equivalence relation is a monoïd congruence

The $k$-type of $u \cdot v$ is determined by the $k$-types of $u$ and $v
Remark
- The number of MSO sentences of quantifier depth at most $k$ is finite
- We define an equivalence relation over words
  - Two words are $k$-equivalent iff no formula of q.d. $k$ can distinguish them
  - This equivalence relation has finite index
  - We denote $k$-types these equivalence classes

Remark: This equivalence relation is a monoïd congruence
The $k$-type of $u \cdot v$ is determined by the $k$-types of $u$ and $v$

Remark
Formulas with quantifier depth $k$ and 2 first-order free variables:
$$
\varphi(x, y) \quad w : \quad \overline{w_1} \quad x \quad \overline{w_2} \quad y \quad \overline{w_3}
$$
Finiteness of MSO formula up to some quantifier depth

Remark
- The number of MSO sentences of quantifier depth at most $k$ is finite
- We define an equivalence relation over words
  - Two words are $k$-equivalent iff no formula of q.d. $k$ can distinguish them
  - This equivalence relation has finite index
  - We denote $k$-types these equivalence classes

Remark: This equivalence relation is a monoïd congruence
The $k$-type of $u \cdot v$ is determined by the $k$-types of $u$ and $v$

Remark
Formulas with quantifier depth $k$ and 2 first-order free variables:

$$
\varphi(x, y) \\
\begin{array}{llllll}
  w : & w_1 & x & w_2 & y & w_3
\end{array}
$$

The validity of $\varphi$ only depends on $w[x]$, $w[y]$ and the $k$-types of $w_1$, $w_2$, $w_3$
A crossing at position $x$ is an edge which connects two nodes which are not on the same side w.r.t. $x$. 
Boundedly many crossings

Theorem

At any given position there are at most $2C \cdot |k\text{-types}|$ crossings

Otherwise in the image, two distinct nodes have an outgoing edge to the same node
Boundedly many crossings means boundedly many registers

\[ w : \quad x \]

One register for each triple \( k \)-type, letter, \( k \)-type would be enough
Boundedly many crossings means boundedly many registers

\[ w : \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad x - ? - ? - ? - ? - ? - ? - ? - ? - ? - ? \]

- One register for each triple \( k \)-type, letter, \( k \)-type would be enough
- We will also need to handle all possible behaviours
Handling all possible behaviours with Regular Look-Ahead

- Regular look-ahead: guards on transitions (and updates)
  MSO queries over the suffix.

Is there in the image some subword that starts at a position \( y \) (labeled by \( \alpha \)), before \( x \) such that the \( k \)-type of \( w[0:y] \) is \( \tau_1 \) and the \( k \)-type of \( w(y:x) \) is \( \tau_2 \)?

This is an MSO query with quantifier depth \( K = k + |C| + 3 \)!

Thus all the possible cases are handled by "guessing" the \( K \)-type of \( w(x:|w|) \)

The set of registers will be \( k \)-types \( \times \Sigma \times k \)-types \( \times K \)-types

The set of states will be the set of \( K \)-types (the state will state which is the \( K \)-type of the prefix read so far).
Handling all possible behaviours with Regular Look-Ahead

- Regular look-ahead: guards on transitions (and updates)
  MSO queries over the suffix.

\[ w : \quad \underline{\phantom{y}} \quad y \quad \underline{\phantom{x}} \quad \underline{\phantom{-?\quad -?\quad -?\quad -?\quad -?\quad -?\quad -?\quad -?\quad -?\quad -?\quad -?\quad -?\quad -?}} \]

- Is there in the image some subword that starts at a position \( y \) (labeled by \( \alpha \)), before \( x \) such that the \( k \)-type of \( w[0:y) \) is \( \tau_1 \) and the \( k \)-type of \( w(y:x) \) is \( \tau_2 \)?
Handling all possible behaviours with Regular Look-Ahead

- Regular look-ahead: guards on transitions (and updates)
  MSO queries over the suffix.

\[
w : \quad y \quad x \quad ? \quad ? \quad ? \quad ? \quad ? \quad ? \quad ?
\]

- Is there in the image some subword that starts at a position \( y \) (labeled by \( \alpha \)), before \( x \) such that the \( k \)-type of \( w[0:y] \) is \( \tau_1 \) and the \( k \)-type of \( w(y:x) \) is \( \tau_2 \)?

- This is an MSO query with quantifier depth \( K = k + |C| + 3 \).
Handling all possible behaviours with Regular Look-Ahead

- Regular look-ahead: guards on transitions (and updates) MSO queries over the suffix.


- Is there in the image some subword that starts at a position \( y \) (labeled by \( \alpha \)), before \( x \) such that the \( k \)-type of \( w[0:y] \) is \( \tau_1 \) and the \( k \)-type of \( w(y:x) \) is \( \tau_2 \) ?
- This is an MSO query with quantifier depth \( K = k + |C| + 3 \) !
- Thus all the possible cases are handled by “guessing” the \( K \)-type of \( w(x:|w|) \)
Handling all possible behaviours with Regular Look-Ahead

- Regular look-ahead: guards on transitions (and updates) MSO queries over the suffix.

\[ w : \quad \underbrace{\quad y \quad \ldots \quad x \quad \ldots \quad \ldots \quad \ldots \quad \ldots \quad \ldots \quad \ldots \quad \ldots \quad \ldots \quad } \]

- Is there in the image some subword that starts at a position \( y \) (labeled by \( \alpha \)), before \( x \) such that the \( k \)-type of \( w[0:y] \) is \( \tau_1 \) and the \( k \)-type of \( w(y:x) \) is \( \tau_2 \) ?
- This is an MSO query with quantifier depth \( K = k + |C| + 3 \) !
- Thus all the possible cases are handled by “guessing” the \( K \)-type of \( w(x:|w|) \)
- The set of registers will be \( k \text{-types } \times \Sigma \times k \text{-types } \times K \text{-types} \)
- The set of states will be the set of \( K \)-types (the state will state which is the \( K \)-type of the prefix read so far).
Finite word case

- At the end of the input, we output the non-empty register corresponding to the regular-look ahead $\varepsilon$. 
Finite word case

- At the end of the input, we output the non-empty register corresponding to the regular-look ahead $\varepsilon$.
- The reduction does not go through a two-way model.
Infinite word input case

- The main difficulty lies in that we have to guess correctly and infinitely often the regular look-ahead.
Infinite word input case

- The main difficulty lies in that we have to guess correctly and infinitely often the regular look-ahead.
- The image is defined as the limit of the content of some register, depending on the set of infinitely occurring states (Muller condition).
Infinite word input case

- The main difficulty lies in that we have to guess correctly and infinitely often the regular look-ahead.
- The image is defined as the limit of the content of some register, depending on the set of infinitely occurring states (Muller condition).
- With the Muller output condition, we can have some MSO property over the whole word.
Infinite word input case

- The main difficulty lies in that we have to guess correctly and infinitely often the regular look-ahead.
- The image is defined as the limit of the content of some register, depending on the set of infinitely occurring states (Muller condition).
- With the Muller output condition, we can have some MSO property over the whole word.
- We need to effectively find a factorization $\tau(\tau')^\omega$ of the input. This can be found in Shelah’s alternative proof of Büchi Theorem, using a finite additive coloring (Ramsey’s Theorem).
Infinite word input case

- The main difficulty lies in that we have to guess correctly and infinitely often the regular look-ahead.
- The image is defined as the limit of the content of some register, depending on the set of infinitely occurring states (Muller condition)
- With the Muller output condition, we can have some MSO property over the whole word.
- We need to effectively find a factorization $\tau(\tau')^\omega$ of the input. This can be found in Shelah’s alternative proof of Büchi Theorem, using a finite additive coloring (Ramsey’s Theorem)
- Thus we can output infinitely often some increasing prefixes of the image
Infinite word input case

- The main difficulty lies in that we have to guess correctly and infinitely often the regular look-ahead.
- The image is defined as the limit of the content of some register, depending on the set of infinitely occurring states (Muller condition)
- With the Muller output condition, we can have some MSO property over the whole word.
- We need to effectively find a factorization $\tau(\tau')^\omega$ of the input. This can be found in Shelah’s alternative proof of Büchi Theorem, using a finite additive coloring (Ramsey’s Theorem)
- Thus we can output infinitely often some increasing prefixes of the image
- We converge toward the output
Contents

1 Regular Transformations
   • Logical definition
   • Streaming Transducers

2 Contribution: Equivalence with a direct logic-based reduction
   • Some logical considerations
   • Proof Walkthrough

3 Decision procedures
   • Functional equivalence
   • Typechecking Problem

4 Conclusion
Functional equivalence is decidable

Do two transformations have the same image on any input?

A hard problem: different logical ways to define the same transformation

\[
\begin{align*}
    a_0 & \quad a_1 & \quad a_2 & \ldots & a_n - 1 & \quad a_n \\
    o_1 & \quad a_1 & \quad a_2 & \ldots & a_n - 1 & \quad a_n \\
\end{align*}
\]

Reduction to reachability in a counter system (no test, no decrement):

▶ Idea: finding a conflicting position (say \( a \) in first image, \( b \) in the second)
▶ Two counters tracking the number of letters before the conflicting position in each image
▶ Set of states: (states of the transducer \( \times \) 4 registers of the transducer)

\( \text{⋆}\text{0: the value of this register does not appear in the output} \)
\( \text{⋆}\text{1: its value appears before the conflicting position} \)
\( \text{⋆}\text{2: its value contains the conflicting position} \)
\( \text{⋆}\text{3: its value is after the conflicting position} \)

▶ Erase the letters in the transitions, increment corresponding to the registers updates
▶ Find a reachable configuration where the two counters are equal
Functional equivalence is decidable

Do two transformations have the same image on any input?

- A hard problem: different logical ways to define the same transformation
  \[ w : \quad a_0 \rightarrow a_1 \rightarrow a_2 \rightarrow \ldots \rightarrow a_{n-1} \rightarrow a_n \]
  \[ o_1 : \quad a_1 \rightarrow a_2 \rightarrow \ldots \rightarrow a_{n-1} \rightarrow a_n \]
  \[ o_2 : \quad a_1 \rightarrow a_2 \rightarrow a_3 \rightarrow \ldots \rightarrow a_n \]
Functional equivalence is decidable

Do two transformations have the same image on any input?

- A hard problem: different logical ways to define the same transformation
  \[ w : a_0 \rightarrow a_1 \rightarrow a_2 \rightarrow \ldots \rightarrow a_{n-1} \rightarrow a_n \]
  \[ o_1 : a_1 \rightarrow a_2 \rightarrow \ldots \rightarrow a_{n-1} \rightarrow a_n \]
  \[ o_2 : a_1 \rightarrow a_2 \rightarrow a_3 \rightarrow \ldots \rightarrow a_n \]

- Reduction to reachability in a counter system (no test, no decrement):
  - Idea: finding a conflicting position (say \( a \) in first image, \( b \) in the second)
  - Two counters tracking the number of letters before the conflicting position in each image
  - Set of states: \((\text{states of the transducer} \times 4^{\text{registers of the transducer}})^2\)
    - 0: the value of this register does not appear in the output
    - 1: its value appears before the conflicting position
    - 2: its value contains the conflicting position
    - 3: its value is after the conflicting position
  - Erase the letters in the transitions, increment corresponding to the registers updates
  - Find a reachable configuration where the two counters are equal
The typechecking problem

**Definition**

Given formulas $\varphi, \psi$ do we have $\forall w. w \models \varphi \implies T(w) \models \psi$

- We can perform this check by some automatic construction
Contents

1. Regular Transformations
   • Logical definition
   • Streaming Transducers

2. Contribution: Equivalence with a direct logic-based reduction
   • Some logical considerations
   • Proof Walkthrough

3. Decision procedures
   • Functional equivalence
   • Typechecking Problem

4. Conclusion
Conclusion

Contributions

- Direct proof of equivalence between Courcelle transformations and Streaming Transducers
- Previously unexplored setting of $\omega$-words to trees
- Syntactically ensuring convergence of the output
- Equivalence and type-checking problems are decidable

Perspectives

- More expressive transformations (relaxing the restriction on copies)
- Less expressive transformations (First-order fragment)
- Extension to regular cost functions
Conclusion

Contributions

- Direct proof of equivalence between Courcelle transformations and Streaming Transducers
- Previously unexplored setting of $\omega$-words to trees
- Syntactically ensuring convergence of the output
- Equivalence and type-checking problems are decidable

Perspectives

- More expressive transformations (relaxing the restriction on copies)
- Less expressive transformations (First-order fragment)
- Extension to regular cost functions

Thank you for your attention!


Weak second-order Arithmetic and Finite Automata.  

Serial composition of 2-way finite-state transducers and simple programs on strings.  

Monadic second-order definable graph transductions: a survey.  

MSO definable string transductions and two-way finite-state transducers.  
Alternating finite automata on omega-words.