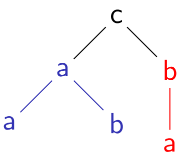
Stackless Processing of Streamed Trees

Corentin Barloy, Filip Murlak, Charles Paperman

LINKS seminar

Processing streamed trees

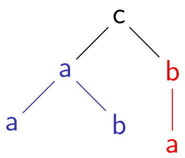
XML encoding of trees:



$\langle c\rangle$
$\quad\langle a\rangle$
$\quad\quad\langle a\rangle\langle/a\rangle$
$\quad\quad\langle b\rangle\langle/b\rangle$
$\quad\langle/a\rangle$
$\quad\langle b\rangle$
$\quad\quad\langle a\rangle\langle/a\rangle$
$\quad\langle/b\rangle$
$\langle/c\rangle$

Processing streamed trees

XML encoding of trees:



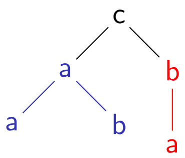
$\langle c \rangle$
$\quad \langle a \rangle$
$\quad\quad \langle a \rangle \langle /a \rangle$
$\quad\quad \langle b \rangle \langle /b \rangle$
$\quad \langle /a \rangle$
$\quad \langle b \rangle$
$\quad\quad \langle a \rangle \langle /a \rangle$
$\quad \langle /b \rangle$
$\langle /c \rangle$

Two problems:

- validation
- querying

Processing streamed trees

XML encoding of trees:



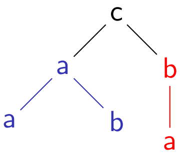
$\langle c \rangle$
$\quad \langle a \rangle$
$\qquad \langle a \rangle \langle / a \rangle$
$\qquad \langle b \rangle \langle / b \rangle$
$\quad \langle / a \rangle$
$\quad \langle b \rangle$
$\qquad \langle a \rangle \langle / a \rangle$
$\quad \langle / b \rangle$
$\langle / c \rangle$

Two problems:

- validation - decide whether a tree complies with a given specification.
- querying

Processing streamed trees

XML encoding of trees:



$\langle c \rangle$
$\quad \langle a \rangle$
$\quad\quad \langle a \rangle \langle /a \rangle$
$\quad\quad \langle b \rangle \langle /b \rangle$
$\quad \langle /a \rangle$
$\quad \langle b \rangle$
$\quad\quad \langle a \rangle \langle /a \rangle$
$\quad \langle /b \rangle$
$\langle /c \rangle$

Two problems:

- validation - decide whether a tree complies with a given specification.
- querying - select all nodes satisfying a given query.

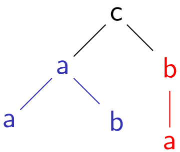
Querying

RPQs: the path from the root belongs to a given regular language.

Querying

RPQs: the path from the root belongs to a given regular language.

Evaluate the RPQ ca^*b:

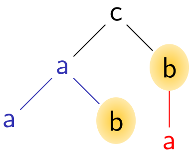


$\langle c \rangle$
$\quad\langle a \rangle$
$\quad\quad\langle a \rangle\langle / a \rangle$
$\quad\quad\langle b \rangle\langle / b \rangle$
$\quad\langle / a \rangle$
$\quad\langle b \rangle$
$\quad\quad\langle a \rangle\langle / a \rangle$
$\quad\langle / b \rangle$
$\langle / c \rangle$

Querying

RPQs: the path from the root belongs to a given regular language.

Evaluate the RPQ ca^*b:

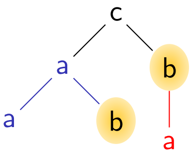


$\langle c \rangle$
$\langle a \rangle$
$\langle a \rangle \langle /a \rangle$
$\langle b \rangle \langle /b \rangle$
$\langle /a \rangle$
$\langle b \rangle$
$\langle a \rangle \langle /a \rangle$
$\langle /b \rangle$
$\langle /c \rangle$

Querying

RPQs: the path from the root belongs to a given regular language.

Evaluate the RPQ ca^*b:



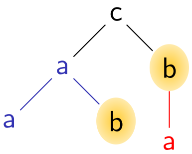
$\langle c \rangle$
$\langle a \rangle$
$\langle a \rangle \langle / a \rangle$
$\langle b \rangle \langle / b \rangle$
$\langle / a \rangle$
$\langle b \rangle$
$\langle a \rangle \langle / a \rangle$
$\langle / b \rangle$
$\langle / c \rangle$

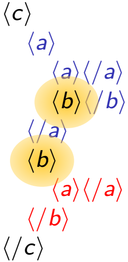
We do pre-selecting for more flexibility.

Querying

RPQs: the path from the root belongs to a given regular language.

Evaluate the RPQ ca^*b:





We do pre-selecting for more flexibility.

Motivation: Path Queries = Order-invariant Queries.

Evaluation in constant memory

Which RPQs can be evaluated in constant memory?

Theorem (Effective characterisation)

RPQ L can be evaluated in constant memory $\Leftrightarrow$ L is (almost) reversible

Which RPQs can be evaluated in constant memory?

Theorem (Effective characterisation)

RPQ L can be evaluated in constant memory $\Leftrightarrow$ L is (almost) reversible

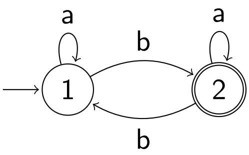
The minimal automaton is co-deterministic, that is, after reversing the arrows it is deterministic.

Which RPQs can be evaluated in constant memory?

Theorem (Effective characterisation)

RPQ L can be evaluated in constant memory $\Leftrightarrow$ L is (almost) reversible

The minimal automaton is co-deterministic, that is, after reversing the arrows it is deterministic.



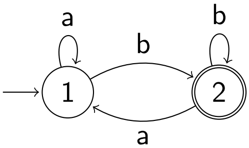
$(a^* ba^* ba^*)^*$ is doable.

Which RPQs can be evaluated in constant memory?

Theorem (Effective characterisation)

RPQ L can be evaluated in constant memory $\Leftrightarrow$ L is (almost) reversible

The minimal automaton is co-deterministic, that is, after reversing the arrows it is deterministic.



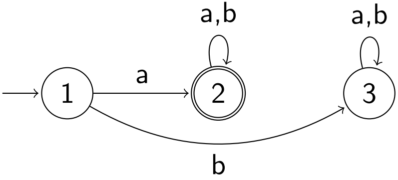
We can check last letters for free: $\begin{matrix} \{a,b\}^* b \\ //b \end{matrix}$ is doable.

Which RPQs can be evaluated in constant memory?

Theorem (Effective characterisation)

RPQ L can be evaluated in constant memory $\Leftrightarrow$ L is (almost) reversible

The minimal automaton is co-deterministic, that is, after reversing the arrows it is deterministic.



We can check the root for free: $\begin{matrix} a\{a,b\}^* \\ a// \end{matrix}$ is doable.

Proof sketch

$\Leftarrow$: If L is reversible then it can be evaluated in constant memory.

Proof sketch

$\Leftarrow$: If L is reversible then it can be evaluated in constant memory.

Algorithm:

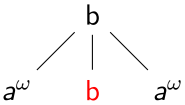
- When an opening tag is read, follow the transition in the automaton.
- When a closing tag is read, follow the transition in the reverse automaton.

Proof sketch

$\Rightarrow$: $\begin{array}{c}(a+b+c)^*ab\\ //a/b\end{array}$ cannot be evaluated in constant memory.

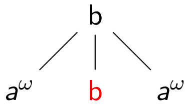
Proof sketch

$\Rightarrow$: $\begin{array}{c}(a+b+c)^*ab \\ //a/b\end{array}$ cannot be evaluated in constant memory.



Proof sketch

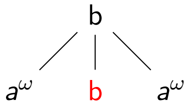
$\Rightarrow$: $\begin{array}{c}(a+b+c)^*ab\\ //a/b\end{array}$ cannot be evaluated in constant memory.



$$\langle b\rangle \quad \langle a\rangle\cdots\langle a\rangle\langle /a\rangle\cdots\langle /a\rangle \quad \langle b\rangle\langle /b\rangle \quad \langle a\rangle\cdots\langle a\rangle\langle /a\rangle\cdots\langle /a\rangle \quad \langle /b\rangle$$

Proof sketch

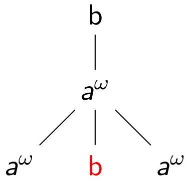
$\Rightarrow$: $\begin{array}{c}(a+b+c)^*ab \\ //a/b\end{array}$ cannot be evaluated in constant memory.



$$\langle b\rangle \quad \langle a\rangle\cdots\langle a\rangle\langle /a\rangle\cdots\langle /a\rangle \quad \langle b\rangle\langle /b\rangle \quad \langle a\rangle\cdots\langle a\rangle\langle /a\rangle\cdots\langle /a\rangle \quad \langle /b\rangle$$

Proof sketch

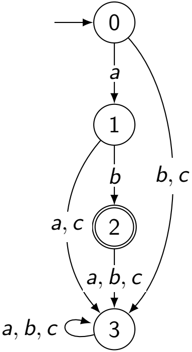
$\Rightarrow$: $\begin{matrix}(a+b+c)^*ab \\ //a/b\end{matrix}$ cannot be evaluated in constant memory.



$$\langle b\rangle \quad \langle a\rangle \cdots \langle a\rangle\langle /a\rangle \cdots \langle /a\rangle \quad \langle b\rangle\langle /b\rangle \quad \langle a\rangle \cdots \langle a\rangle\langle /a\rangle \cdots \langle /a\rangle \quad \langle /b\rangle$$

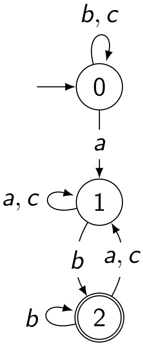
Limitations of constant memory evaluation

Many queries cannot be evaluated in constant memory.



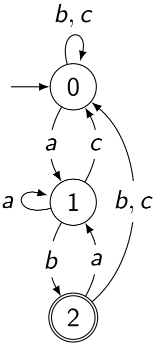
ab

$/a/b$



$(a+b+c)^*a(a+b+c)^*b$

$//a//b$

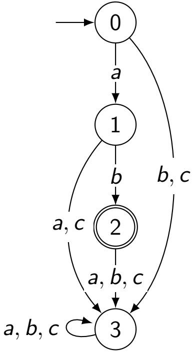


$(a+b+c)^*ab$

$//a/b$

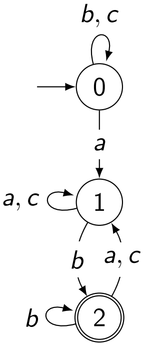
Limitations of constant memory evaluation

Many queries cannot be evaluated in constant memory.



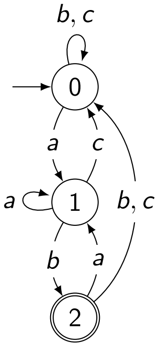
ab

$/a/b$



$(a+b+c)^*a(a+b+c)^*b$

$//a//b$



$(a+b+c)^*ab$

$//a/b$

They can be evaluated using a stack, but this is costly (memory linear in the depth).

Stackless queries
(Evaluation in logarithmic memory)

Stackless automata

Main ingredients:
- a finite state machine,
- a counter that stores the current depth in the tree,
- a finite number of registers where the counter values can be stored,
- can compare register values with the current depth.

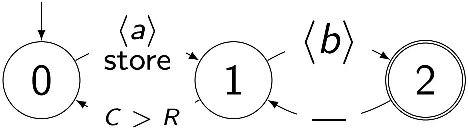
Stackless automata

Main ingredients:
- a finite state machine,
- a counter that stores the current depth in the tree,
- a finite number of registers where the counter values can be stored,
- can compare register values with the current depth.

Evaluating:

$(a+b+c)^*a(a+b+c)^*b$

//a//b



Effective characterisation of stackless RPQs

Theorem

The RPQ L is stackless $\Leftrightarrow$ L is hierarchically (almost) reversible

Effective characterisation of stackless RPQs

Theorem

The RPQ L is stackless $\Leftrightarrow$ L is hierarchically (almost) reversible

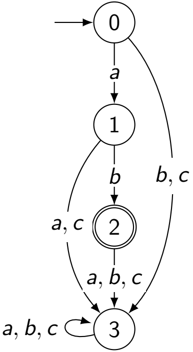
Each strongly connected component is (almost) reversible:

Effective characterisation of stackless RPQs

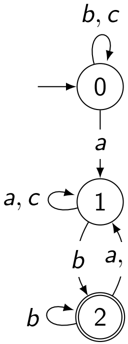
Theorem

The RPQ L is stackless $\Leftrightarrow$ L is hierarchically (almost) reversible

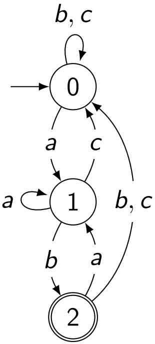
Each strongly connected component is (almost) reversible:



$/a/b$
is HAR



$//a//b$
is HAR



$//a/b$
is not HAR

Proof sketch

$\Leftarrow$: If L is hierarchically reversible then it can be evaluated stacklessly.

Proof sketch

$\Leftarrow$: If L is hierarchically reversible then it can be evaluated stacklessly.

Algorithm:

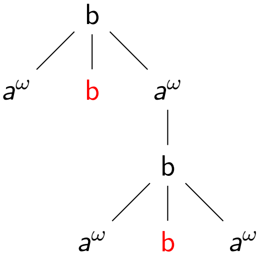
- When an opening tag is read, follow the transition in the automaton.
  If there is a change of SCC, store the depth in a register.
- When a closing tag is read, follow the transition in the reverse automaton.
  If the depth reached is stored, resume the computation in the previous SCC instead.

Proof sketch

$\Rightarrow$: $\begin{array}{c}(a+b+c)^*ab\\ //a/b\end{array}$ cannot be evaluated stacklessly.

Proof sketch

$\Rightarrow$: $\begin{array}{c}(a+b+c)^* ab \\ //a/b\end{array}$ cannot be evaluated stacklessly.



Expressivity outside of RPQs

There are stackless languages that are not regular:

The set of trees such that all a's are at the same depth.

Expressivity outside of RPQs

There are stackless languages that are not regular:
The set of trees such that all a's are at the same depth.

We can ensure regularity by restricting the use of the registers.

Expressivity outside of RPQs

There are stackless languages that are not regular:
The set of trees such that all a's are at the same depth.

We can ensure regularity by restricting the use of the registers.

Trees with a given descendant pattern is stackless.

Expressivity outside of RPQs

There are stackless languages that are not regular:
The set of trees such that all a's are at the same depth.

We can ensure regularity by restricting the use of the registers.

Trees with a given descendant pattern is stackless.

Finding a sequence of consecutive siblings is not stackless.

When can we validate a document in constant memory?

Validation

Check if the tree conforms to the given schema, modelled as a regular tree language.

When can we validate a document in constant memory?

Validation

Check if the tree conforms to the given schema, modelled as a regular tree language.

Weak validation [Segoufin, Vianu PODS'02]

Assume that the input is a correct encoding of some tree.

When can we validate a document in constant memory?

Validation

Check if the tree conforms to the given schema, modelled as a regular tree language.

Weak validation [Segoufin, Vianu PODS'02]

Assume that the input is a correct encoding of some tree.

Characterize the schemas that can be weakly validated in constant memory.

When can we validate a document in constant memory?

Validation

Check if the tree conforms to the given schema, modelled as a regular tree language.

Weak validation [Segoufin, Vianu PODS'02]

Assume that the input is a correct encoding of some tree.

Characterize the schemas that can be weakly validated in constant memory.

Segoufin & Vianu solved it for fully recursive DTDs.

We solve it for tree languages of the form: each branch is in language L.

General problem still open, both for constant-memory and our stackless model.

Conclusion

- Similar characterisations hold for JSON-like encoding, where closing tags carry no information on the letters.

Conclusion

- Similar characterisations hold for JSON-like encoding, where closing tags carry no information on the letters.
- Ongoing work on leveraging schemas for querying streamed trees.

Conclusion

- Similar characterisations hold for JSON-like encoding, where closing tags carry no information on the letters.
- Ongoing work on leveraging schemas for querying streamed trees.
- Ongoing work on vectorization.