Regular XPath: Algebra, Logic and Automata

Balder ten Cate

Szeged, 1 October 2006

The topic of this talk

- This talk is about languages for describing binary relations in trees.
- Binary relations means that
  - Instead of sentences, we use formulas with two free variables $\phi(x, y)$
  - Our tree walking automata can start and finish their walk anywhere in the tree, not neccessarily at the root.
- The motivation comes from XML
- Specifically, we are interested in the XML path language Regular XPath.
- We would like to characterize this language in terms of logic and/or automata.

The topic of this talk

- This talk is about languages for describing binary relations in trees.
- Binary relations means that
  - Instead of sentences, we use formulas with two free variables $\phi(x, y)$
  - Our tree walking automata can start and finish their walk anywhere in the tree, not neccessarily at the root.
- The motivation comes from XML
- Specifically, we are interested in the XML path language Regular XPath.
- We would like to characterize this language in terms of logic and/or automata.

The topic of this talk

- This talk is about languages for describing binary relations in trees.
- Binary relations means that
  - Instead of sentences, we use formulas with two free variables $\phi(x, y)$
  - Our tree walking automata can start and finish their walk anywhere in the tree, not neccessarily at the root.
- The motivation comes from XML
- Specifically, we are interested in the XML path language Regular XPath.
- We would like to characterize this language in terms of logic and/or automata.

The topic of this talk

- This talk is about languages for describing binary relations in trees.
- Binary relations means that
  - Instead of sentences, we use formulas with two free variables $\phi(x, y)$
  - Our tree walking automata can start and finish their walk anywhere in the tree, not neccessarily at the root.
- The motivation comes from XML
- Specifically, we are interested in the XML path language Regular XPath.
- We would like to characterize this language in terms of logic and/or automata.

The topic of this talk

- This talk is about languages for describing binary relations in trees.
- Binary relations means that
  - Instead of sentences, we use formulas with two free variables $\phi(x, y)$
  - Our tree walking automata can start and finish their walk anywhere in the tree, not neccessarily at the root.
- The motivation comes from XML
- Specifically, we are interested in the XML path language Regular XPath.
- We would like to characterize this language in terms of logic and/or automata.

The topic of this talk

- This talk is about languages for describing binary relations in trees.
- Binary relations means that
  - Instead of sentences, we use formulas with two free variables $\phi(x, y)$
  - Our tree walking automata can start and finish their walk anywhere in the tree, not neccessarily at the root.
- The motivation comes from XML
- Specifically, we are interested in the XML path language Regular XPath.
- We would like to characterize this language in terms of logic and/or automata.

The topic of this talk

- This talk is about languages for describing binary relations in trees.
- Binary relations means that
  - Instead of sentences, we use formulas with two free variables $\phi(x, y)$
  - Our tree walking automata can start and finish their walk anywhere in the tree, not neccessarily at the root.
- The motivation comes from XML
- Specifically, we are interested in the XML path language Regular XPath.
- We would like to characterize this language in terms of logic and/or automata.

XML documents

- XML documents are (for present purposes) are finite unranked sibling-ordered node labelled trees.
- So, an XML document is a tuple $T = (N, R_{\downarrow}, R_{\rightarrow}, V)$ where
  - N is the set of nodes,
  - $R_{\downarrow}$ and $R_{\rightarrow}$ are the 'child' and 'next sibling' relations, and
  - $V : N \rightarrow \Sigma$.

XML documents

- XML documents are (for present purposes) are finite unranked sibling-ordered node labelled trees.
- So, an XML document is a tuple $T = (N, R_\downarrow, R_\rightarrow, V)$ where
  - N is the set of nodes,
  - $R_\downarrow$ and $R_\rightarrow$ are the 'child' and 'next sibling' relations, and
  - $V : N \rightarrow \Sigma$.

XML documents

- XML documents are (for present purposes) are finite unranked sibling-ordered node labelled trees.
- So, an XML document is a tuple $T = (N, R_{\downarrow}, R_{\rightarrow}, V)$ where
  - N is the set of nodes,
  - $R_{\downarrow}$ and $R_{\rightarrow}$ are the ‘child’ and ‘next sibling’ relations, and
  - $V : N \rightarrow \Sigma$.

XML documents

- XML documents are (for present purposes) are finite unranked sibling-ordered node labelled trees.
- So, an XML document is a tuple $T = (N, R_\downarrow, R_\rightarrow, V)$ where
  - N is the set of nodes,
  - $R_\downarrow$ and $R_\rightarrow$ are the 'child' and 'next sibling' relations, and
  - $V : N \rightarrow \Sigma$.

XML documents

- XML documents are (for present purposes) are finite unranked sibling-ordered node labelled trees.
- So, an XML document is a tuple $T = (N, R_{\downarrow}, R_{\rightarrow}, V)$ where
  - N is the set of nodes,
  - $R_{\downarrow}$ and $R_{\rightarrow}$ are the 'child' and 'next sibling' relations, and
  - $V : N \rightarrow \Sigma$.

Syntax of Regular XPath

- Regular XPath has two types of expressions:
  - path expressions
    $\alpha \ ::= \uparrow \mid \downarrow \mid \leftarrow \mid \rightarrow \mid . \mid \alpha/\beta \mid \alpha \cup \beta \mid \alpha^* \mid \alpha[\phi]$
  - node expressions
    $\phi \ ::= \ p \mid \neg\phi \mid \phi \wedge \psi \mid \langle\alpha\rangle$
- Path expression define binary relations. When applied to a given "context node", they yield a set of nodes.

  Node expressions define sets of nodes.
- We use $/\alpha$ as shorthand for $\uparrow^* [\neg\langle \uparrow \rangle]/\alpha$.

Syntax of Regular XPath

- Regular XPath has two types of expressions:
  - path expressions
    $\alpha \ ::= \uparrow \mid \downarrow \mid \leftarrow \mid \rightarrow \mid \ . \mid \alpha/\beta \mid \alpha \cup \beta \mid \alpha^* \mid \alpha[\phi]$
  - node expressions
    $\phi \ ::= \ p \mid \neg\phi \mid \phi \wedge \psi \mid \langle\alpha\rangle$
- Path expression define binary relations. When applied to a given "context node", they yield a set of nodes.
  Node expressions define sets of nodes.
- We use $/\alpha$ as shorthand for $\uparrow^* [\neg\langle \uparrow \rangle]/\alpha$.

Syntax of Regular XPath

- Regular XPath has two types of expressions:
  - path expressions
    $\alpha ::= \uparrow \mid \downarrow \mid \leftarrow \mid \rightarrow \mid . \mid \alpha/\beta \mid \alpha \cup \beta \mid \alpha^* \mid \alpha[\phi]$
  - node expressions
    $\phi ::= p \mid \neg\phi \mid \phi \wedge \psi \mid \langle\alpha\rangle$

- Path expression define binary relations. When applied to a given "context node", they yield a set of nodes.
  Node expressions define sets of nodes.

- We use $/\alpha$ as shorthand for $\uparrow^* [\neg\langle \uparrow \rangle]/\alpha$.

Syntax of Regular XPath

- Regular XPath has two types of expressions:
  - path expressions
    $\alpha \ ::= \uparrow \mid \downarrow \mid \leftarrow \mid \rightarrow \mid \ . \ \mid \alpha/\beta \mid \alpha \cup \beta \mid \alpha^* \mid \alpha[\phi]$
  - node expressions
    $\phi \ ::= \ p \mid \neg\phi \mid \phi \wedge \psi \mid \langle\alpha\rangle$
- Path expression define binary relations. When applied to a given "context node", they yield a set of nodes.

  Node expressions define sets of nodes.
- We use $/\alpha$ as shorthand for $\uparrow^* [\neg\langle \uparrow \rangle]/\alpha$.

Syntax of Regular XPath

- Regular XPath has two types of expressions:
  - path expressions
    $\alpha \ ::= \uparrow \mid \downarrow \mid \leftarrow \mid \rightarrow \mid . \mid \alpha/\beta \mid \alpha \cup \beta \mid \alpha^* \mid \alpha[\phi]$
  - node expressions
    $\phi \ ::= \ p \mid \neg\phi \mid \phi \wedge \psi \mid \langle\alpha\rangle$

- Path expression define binary relations. When applied to a given "context node", they yield a set of nodes.

  Node expressions define sets of nodes.

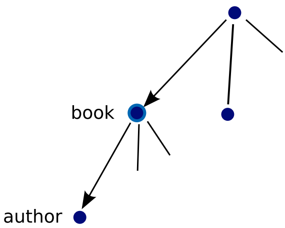
- We use $/\alpha$ as shorthand for $\uparrow^* [\neg\langle \uparrow \rangle]/\alpha$.

Semantics of Regular XPath

$$
\begin{aligned}
[\![\alpha]\!]^M &= R_\alpha \qquad \text{for } \alpha \in \{\downarrow, \uparrow, \leftarrow, \rightarrow\} \\
[\![.]\!]^M &= \text{the identity relation on } N \\
[\![\alpha/\beta]\!]^M &= \text{composition of } [\![\alpha]\!]^M \text{ and } [\![\beta]\!]^M \\
[\![\alpha \cup \beta]\!]^M &= \text{union of } [\![\alpha]\!]^M \text{ and } [\![\beta]\!]^M \\
[\![\alpha^*]\!]^M &= \text{reflexive transitive closure of } [\![\alpha]\!]^M \\
[\![\alpha[\phi]]\!]^M &= \{(n, m) \in [\![\alpha]\!]^M \mid m \in [\![\phi]\!]^M\}
\end{aligned}
$$

$$
\begin{aligned}
[\![p]\!]^M &= V(p) \\
[\![\phi \wedge \psi]\!]^M &= [\![\phi]\!]^M \cap [\![\psi]\!]^M \\
[\![\neg\phi]\!]^M &= N \setminus [\![\phi]\!]^M \\
[\![\langle\alpha\rangle]\!]^M &= \text{domain of } [\![\alpha]\!]^M = \{n \mid (n, m) \in [\![\alpha]\!]^M\}
\end{aligned}
$$

An example



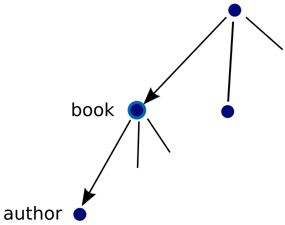
"Go to the next book that has at least two authors."

In Regular XPath:

$$(\rightarrow [\neg \mathit{twoauthorbook}])^* / \rightarrow [\mathit{twoauthorbook}]$$

where *twoauthorbook* stands for
$\mathit{book} \wedge \langle \downarrow [\mathit{author}] / \rightarrow^+ [\mathit{author}] \rangle$.

An example



"Go to the next book that has at least two authors."

In Regular XPath:

$$(\rightarrow [\neg \mathit{twoauthorbook}])^* / \rightarrow [\mathit{twoauthorbook}]$$

where *twoauthorbook* stands for
$\mathit{book} \wedge \langle \downarrow [\mathit{author}] / \rightarrow^+ [\mathit{author}] \rangle$.

The following can be expressed in Regular XPath:

> *"Go to the root if it has an even number of descendants, otherwise retrieve nothing"*

To see this, note that

- Let $(\alpha \;\; \texttt{while} \;\; \phi)$ be shorthand for $(.[\phi]/\alpha)^*$
- Let $\texttt{suc}$ be shorthand for $\downarrow[\neg\langle\leftarrow\rangle] \;\cup\; .[\neg\langle\downarrow\rangle]/(\uparrow \texttt{while} \; \neg\langle\rightarrow\rangle)/\rightarrow$ (the successor in depth first left-to-right ordering).
- Then $/(\texttt{suc}/\texttt{suc})^*[\neg\langle\downarrow\rangle]/(\uparrow \texttt{while} \; \neg\langle\rightarrow\rangle)[\neg\langle\uparrow\rangle]$ expresses the intended query.

Another example

The following can be expressed in Regular XPath:

"Go to the root if it has an even number of descendants, otherwise retrieve nothing"

To see this, note that

- Let $(\alpha\ \texttt{while}\ \phi)$ be shorthand for $(.[\phi]/\alpha)^*$
- Let $\texttt{suc}$ be shorthand for $\downarrow[\neg\langle\leftarrow\rangle]\ \cup\ .[\neg\langle\downarrow\rangle]/(\uparrow\ \texttt{while}\ \neg\langle\rightarrow\rangle)/\rightarrow$ (the successor in depth first left-to-right ordering).
- Then $/(\texttt{suc}/\texttt{suc})^*[\neg\langle\downarrow\rangle]/(\uparrow\ \texttt{while}\ \neg\langle\rightarrow\rangle)[\neg\langle\uparrow\rangle]$ expresses the intended query.

Another example

The following can be expressed in Regular XPath:

"Go to the root if it has an even number of descendants, otherwise retrieve nothing"

To see this, note that

- Let $(\alpha \ \mathtt{while} \ \phi)$ be shorthand for $(.[\phi]/\alpha)^*$
- Let $\mathtt{suc}$ be shorthand for
  $\downarrow[\neg\langle\leftarrow\rangle] \ \cup \ .[\neg\langle\downarrow\rangle]/(\uparrow \mathtt{while} \ \neg\langle\rightarrow\rangle)/\rightarrow$
  (the successor in depth first left-to-right ordering).
- Then $/(\mathtt{suc}/\mathtt{suc})^*[\neg\langle\downarrow\rangle]/(\uparrow \mathtt{while} \ \neg\langle\rightarrow\rangle)[\neg\langle\uparrow\rangle]$
  expresses the intended query.

Another example

The following can be expressed in Regular XPath:

> *"Go to the root if it has an even number of descendants, otherwise retrieve nothing"*

To see this, note that

- Let $(\alpha \ \texttt{while} \ \phi)$ be shorthand for $(.[\phi]/\alpha)^*$
- Let $\texttt{suc}$ be shorthand for
  $\downarrow[\neg\langle\leftarrow\rangle] \ \cup \ .[\neg\langle\downarrow\rangle]/(\uparrow \texttt{while} \ \neg\langle\rightarrow\rangle)/\rightarrow$
  (the successor in depth first left-to-right ordering).
- Then $/(\texttt{suc}/\texttt{suc})^*[\neg\langle\downarrow\rangle]/(\uparrow \texttt{while} \ \neg\langle\rightarrow\rangle)[\neg\langle\uparrow\rangle]$ expresses the intended query.

One more example

- Can we express the following query in Regular XPath?

  "Go to any node with an even number of descendants"

- The previous trick does not work: during the depth-first traversal we might accidentally leave the subtree.
- **Yes, it is possible:** we can count
  - the (parity of the) number of nodes in the entire tree
  - the (parity of the) number of ancestors of a node *n*
  - the (parity of the) number of nodes before *n* in the df order
  - the (parity of the) number of nodes after *n* in the df order (not counting the descendants)
- With a *loop* operator things would be much easier:

$$[\![loop(\alpha)]\!] = \{w \mid (w, w) \in [\![\alpha]\!]\}$$

Using loop we could express it as follows:

$$/\downarrow^* [loop\Big((\mathtt{suc}/\mathtt{suc})^*[\neg\langle\downarrow\rangle]/(\uparrow \ \mathtt{while}\ \neg\langle\rightarrow\rangle)\Big)]$$

One more example

- Can we express the following query in Regular XPath?

  "Go to any node with an even number of descendants"

- The previous trick does not work: during the depth-first traversal we might accidentally leave the subtree.
- **Yes, it is possible:** we can count
  - the (parity of the) number of nodes in the entire tree
  - the (parity of the) number of ancestors of a node n
  - the (parity of the) number of nodes before n in the df order
  - the (parity of the) number of nodes after n in the df order (not counting the descendants)
- With a *loop* operator things would be much easier:

$$[\![loop(\alpha)]\!] = \{w \mid (w, w) \in [\![\alpha]\!]\}$$

Using loop we could express it as follows:

$$/ \downarrow^* [loop\Big((\mathtt{suc}/\mathtt{suc})^*[\neg\langle\downarrow\rangle]/(\uparrow\ \mathtt{while}\ \neg\langle\rightarrow\rangle)\Big)]$$

One more example

- Can we express the following query in Regular XPath?

  "Go to any node with an even number of descendants"

- The previous trick does not work: during the depth-first traversal we might accidentally leave the subtree.
- **Yes, it is possible:** we can count
  - the (parity of the) number of nodes in the entire tree
  - the (parity of the) number of ancestors of a node *n*
  - the (parity of the) number of nodes before *n* in the df order
  - the (parity of the) number of nodes after *n* in the df order (not counting the descendants)
- With a *loop* operator things would be much easier:

  $$[\![loop(\alpha)]\!] = \{w \mid (w, w) \in [\![\alpha]\!]\}$$

  Using loop we could express it as follows:

  $$/\downarrow^* [loop\Big((\texttt{suc}/\texttt{suc})^*[\neg\langle\downarrow\rangle]/(\uparrow \texttt{ while } \neg\langle\rightarrow\rangle)\Big)]$$

One more example

- Can we express the following query in Regular XPath?

  "Go to any node with an even number of descendants"

- The previous trick does not work: during the depth-first traversal we might accidentally leave the subtree.
- **Yes, it is possible:** we can count
  - the (parity of the) number of nodes in the entire tree
  - the (parity of the) number of ancestors of a node n
  - the (parity of the) number of nodes before n in the df order
  - the (parity of the) number of nodes after n in the df order (not counting the descendants)
- With a *loop* operator things would be much easier:

$$[\![loop(\alpha)]\!] = \{w \mid (w, w) \in [\![\alpha]\!]\}$$

Using loop we could express it as follows:

$$/\downarrow^* [loop\Big((\mathtt{suc}/\mathtt{suc})^*[\neg\langle\downarrow\rangle]/(\uparrow \ \mathtt{while} \ \neg\langle\rightarrow\rangle)\Big)]$$

One more example

- Can we express the following query in Regular XPath?

  "Go to any node with an even number of descendants"

- The previous trick does not work: during the depth-first traversal we might accidentally leave the subtree.
- **Yes, it is possible:** we can count
  - the (parity of the) number of nodes in the entire tree
  - the (parity of the) number of ancestors of a node *n*
  - the (parity of the) number of nodes before *n* in the df order
  - the (parity of the) number of nodes after *n* in the df order (not counting the descendants)
- With a *loop* operator things would be much easier:

$$[\![loop(\alpha)]\!] = \{ w \mid (w, w) \in [\![\alpha]\!] \}$$

Using loop we could express it as follows:

$$/ \downarrow^* [loop\Big((\mathtt{suc}/\mathtt{suc})^*[\neg\langle\downarrow\rangle]/(\uparrow\ \mathtt{while}\ \neg\langle\rightarrow\rangle) \Big)]$$

One more example

- Can we express the following query in Regular XPath?

  "Go to any node with an even number of descendants"

- The previous trick does not work: during the depth-first traversal we might accidentally leave the subtree.
- **Yes, it is possible:** we can count
  - the (parity of the) number of nodes in the entire tree
  - the (parity of the) number of ancestors of a node *n*
  - the (parity of the) number of nodes before *n* in the df order
  - the (parity of the) number of nodes after *n* in the df order (not counting the descendants)
- With a *loop* operator things would be much easier:

$$[\![loop(\alpha)]\!] = \{ w \mid (w, w) \in [\![\alpha]\!] \}$$

Using loop we could express it as follows:

$$/ \downarrow^* [loop\Big((\mathtt{suc}/\mathtt{suc})^*[\neg\langle\downarrow\rangle]/(\uparrow\ \mathtt{while}\ \neg\langle\rightarrow\rangle)\Big)]$$

One more example

- Can we express the following query in Regular XPath?

  "Go to any node with an even number of descendants"

- The previous trick does not work: during the depth-first traversal we might accidentally leave the subtree.
- **Yes, it is possible:** we can count
  - the (parity of the) number of nodes in the entire tree
  - the (parity of the) number of ancestors of a node *n*
  - the (parity of the) number of nodes before *n* in the df order
  - the (parity of the) number of nodes after *n* in the df order (not counting the descendants)
- With a *loop* operator things would be much easier:

$$[\![loop(\alpha)]\!] = \{w \mid (w, w) \in [\![\alpha]\!]\}$$

  Using loop we could express it as follows:

$$/ \downarrow^* [loop\Big((\texttt{suc}/\texttt{suc})^*[\neg\langle\downarrow\rangle]/(\uparrow \texttt{ while } \neg\langle\rightarrow\rangle)\Big)]$$

One more example

- Can we express the following query in Regular XPath?

  "Go to any node with an even number of descendants"

- The previous trick does not work: during the depth-first traversal we might accidentally leave the subtree.
- **Yes, it is possible:** we can count
  - the (parity of the) number of nodes in the entire tree
  - the (parity of the) number of ancestors of a node *n*
  - the (parity of the) number of nodes before *n* in the df order
  - the (parity of the) number of nodes after *n* in the df order (not counting the descendants)
- With a *loop* operator things would be much easier:

  $$[\![loop(\alpha)]\!] = \{w \mid (w, w) \in [\![\alpha]\!]\}$$

  Using loop we could express it as follows:

  $$/\downarrow^* [loop\Big((\mathtt{suc}/\mathtt{suc})^*[\neg\langle\downarrow\rangle]/(\uparrow\ \mathtt{while}\ \neg\langle\rightarrow\rangle)\Big)]$$

One more example

- Can we express the following query in Regular XPath?

  "Go to any node with an even number of descendants"

- The previous trick does not work: during the depth-first traversal we might accidentally leave the subtree.
- **Yes, it is possible:** we can count
  - the (parity of the) number of nodes in the entire tree
  - the (parity of the) number of ancestors of a node *n*
  - the (parity of the) number of nodes before *n* in the df order
  - the (parity of the) number of nodes after *n* in the df order (not counting the descendants)
- With a *loop* operator things would be much easier:

  $$[\![loop(\alpha)]\!] = \{ w \mid (w, w) \in [\![\alpha]\!] \}$$

  Using loop we could express it as follows:

  $$/ \downarrow^* [loop\Big((\mathtt{suc}/\mathtt{suc})^*[\neg\langle\downarrow\rangle]/(\uparrow \ \mathtt{while} \ \neg\langle\rightarrow\rangle)\Big)]$$

One more example

- Can we express the following query in Regular XPath?

  "Go to any node with an even number of descendants"

- The previous trick does not work: during the depth-first traversal we might accidentally leave the subtree.
- **Yes, it is possible:** we can count
  - the (parity of the) number of nodes in the entire tree
  - the (parity of the) number of ancestors of a node *n*
  - the (parity of the) number of nodes before *n* in the df order
  - the (parity of the) number of nodes after *n* in the df order (not counting the descendants)
- With a *loop* operator things would be much easier:

$$[\![loop(\alpha)]\!] = \{ w \mid (w, w) \in [\![\alpha]\!] \}$$

Using loop we could express it as follows:

$$/ \downarrow^* [loop\Big((\mathtt{suc}/\mathtt{suc})^*[\neg\langle\downarrow\rangle]/(\uparrow \ \mathtt{while}\ \neg\langle\rightarrow\rangle)\Big)]$$

One more example

- Can we express the following query in Regular XPath?

  "Go to any node with an even number of descendants"

- The previous trick does not work: during the depth-first traversal we might accidentally leave the subtree.
- **Yes, it is possible:** we can count
  - the (parity of the) number of nodes in the entire tree
  - the (parity of the) number of ancestors of a node *n*
  - the (parity of the) number of nodes before *n* in the df order
  - the (parity of the) number of nodes after *n* in the df order (not counting the descendants)
- With a *loop* operator things would be much easier:

  $$[\![loop(\alpha)]\!] = \{w \mid (w, w) \in [\![\alpha]\!]\}$$

  Using loop we could express it as follows:

  $$/ \downarrow^* [loop\Big((\mathtt{suc}/\mathtt{suc})^*[\neg\langle\downarrow\rangle]/(\uparrow \mathtt{~while~} \neg\langle\rightarrow\rangle)\Big)]$$

The main question

- What is the expressive power of Regular XPath?

  I.e., which binary relations are definable by path expressions?

The main question

- What is the expressive power of Regular XPath?

  I.e., which binary relations are definable by path expressions?

An educated guess

- What we know:

$$FO \subsetneq \textit{Regular XPath} \subseteq FO + TC^1$$

  (The first inclusion follows from Marx PODS'04).

- A natural conjecture:

$$\textit{Regular XPath} \equiv FO + TC^1$$

  (after all, Regular XPath has a transitive closure operator!)

- We managed to prove a result along these lines only by extending the language with loop.

An educated guess

- What we know:

  $$FO \subsetneq \textit{Regular XPath} \subseteq FO + TC^1$$

  (The first inclusion follows from Marx PODS'04).

- A natural conjecture:

  $$\textit{Regular XPath} \equiv FO + TC^1$$

  (after all, Regular XPath has a transitive closure operator!)

- We managed to prove a result along these lines only by extending the language with loop.

An educated guess

- What we know:

$$FO \subsetneq \textit{Regular XPath} \subseteq FO + TC^1$$

(The first inclusion follows from Marx PODS'04).

- A natural conjecture:

$$\textit{Regular XPath} \equiv FO + TC^1$$

(after all, Regular XPath has a transitive closure operator!)

- We managed to prove a result along these lines only by extending the language with loop.

More about *loop*

- loop provides a weak form of path intersection: *loop*(α) is equivalent to the node expression $\langle \alpha \cap . \rangle$.
- We denote the extension of Regular XPath with loop by Regular XPath$^{\approx}$.
- Adding loop does not affect the complexity:
  - Query evaluation can still be performed in PTime.
  - Query containment can still be solved in ExpTime.

More about *loop*

- loop provides a weak form of path intersection: *loop*(α) is equivalent to the node expression $\langle \alpha \cap . \rangle$.
- We denote the extension of Regular XPath with loop by Regular XPath$^{\approx}$.
- Adding loop does not affect the complexity:
  - Query evaluation can still be performed in PTime.
  - Query containment can still be solved in ExpTime.

More about *loop*

- loop provides a weak form of path intersection:
  loop(α) is equivalent to the node expression $\langle\alpha \cap .\rangle$.
- We denote the extension of Regular XPath with loop by Regular XPath$^{\approx}$.
- Adding loop does not affect the complexity:
  - Query evaluation can still be performed in PTime.
  - Query containment can still be solved in ExpTime.

More about *loop*

- loop provides a weak form of path intersection:
  loop(α) is equivalent to the node expression $\langle \alpha \cap . \rangle$.
- We denote the extension of Regular XPath with loop by Regular XPath$^{\approx}$.
- Adding loop does not affect the complexity:
  - Query evaluation can still be performed in PTime.
  - Query containment can still be solved in ExpTime.

Main result

- Let $FO + TC^1_{np}$ be the extension of first-order logic with transitive closure over formulas with exactly two free variables.
- $FO + TC^1_{np}$ differs from $FO + TC^1$: the latter has transitive closure over formulas with two designated free variables plus possibly other free variables.

Main result: Regular XPath$^{\approx}$ $\equiv$ FO^1_{np}

More precisely, Regular XPath$^{\approx}$ path expressions define the same binary relations as $FO + TC^1_{np}$ formulas with two free variables.

- **Corollary:** Regular XPath$^{\approx}$ is closed under path intersection and complementation.

Main result

- Let $FO + TC^1_{np}$ be the extension of first-order logic with transitive closure over formulas with exactly two free variables.
- $FO + TC^1_{np}$ differs from $FO + TC^1$: the latter has transitive closure over formulas with two designated free variables plus possibly other free variables.

Main result: Regular XPath$^{\approx}$ $\equiv$ FO^1_{np}

More precisely, Regular XPath$^{\approx}$ path expressions define the same binary relations as $FO + TC^1_{np}$ formulas with two free variables.

- **Corollary:** Regular XPath$^{\approx}$ is closed under path intersection and complementation.

Main result

- Let $FO + TC^1_{np}$ be the extension of first-order logic with transitive closure over formulas with exactly two free variables.
- $FO + TC^1_{np}$ differs from $FO + TC^1$: the latter has transitive closure over formulas with two designated free variables plus possibly other free variables.

Main result: Regular XPath$^{\approx}$ $\equiv$ FO^1_{np}

More precisely, Regular XPath$^{\approx}$ path expressions define the same binary relations as $FO + TC^1_{np}$ formulas with two free variables.

- **Corollary:** Regular XPath$^{\approx}$ is closed under path intersection and complementation.

Main result

- Let $FO + TC^1_{np}$ be the extension of first-order logic with transitive closure over formulas with exactly two free variables.
- $FO + TC^1_{np}$ differs from $FO + TC^1$: the latter has transitive closure over formulas with two designated free variables plus possibly other free variables.

Main result: Regular XPath$^{\approx}$ $\equiv$ FO^1_{np}

More precisely, Regular XPath$^{\approx}$ path expressions define the same binary relations as $FO + TC^1_{np}$ formulas with two free variables.

- **Corollary:** Regular XPath$^{\approx}$ is closed under path intersection and complementation.

Main result

- Let $FO + TC^1_{np}$ be the extension of first-order logic with transitive closure over formulas with exactly two free variables.
- $FO + TC^1_{np}$ differs from $FO + TC^1$: the latter has transitive closure over formulas with two designated free variables plus possibly other free variables.

Main result: Regular XPath$^{\approx}$ $\equiv$ FO^1_{np}

More precisely, Regular XPath$^{\approx}$ path expressions define the same binary relations as $FO + TC^1_{np}$ formulas with two free variables.

- **Corollary:** Regular XPath$^{\approx}$ is closed under path intersection and complementation.

Proof sketch

Difficult direction: $FO + TC^1_{np} \subseteq$ Regular XPath$^{\approx}$

Step 1. Restrict attention to binary branching trees. On such trees $\rightarrow$ can be written as $.[\langle\rightarrow\rangle]/\uparrow/\downarrow[\langle\leftarrow\rangle]$, and likewise for $\leftarrow$. This helps reduce the number of cases.

Step 2. A normal form: a path expression is separated if it is a union of expressions of the form α/β, with α walking upwards and β downwards in the tree.

Step 3. The translation itself from formulas $\phi(x, y) \in FO + TC^1_{np}$ to separated Regular XPath$^{\approx}$ path expressions.
To enable an inductive translation, we use conjunctive tree queries over path expressions.

Crucial lemma: showing that the separated expressions are closed under $*$.

Proof sketch

Difficult direction: $FO + TC^1_{np} \subseteq$ Regular XPath$^{\approx}$

Step 1. Restrict attention to binary branching trees. On such trees $\rightarrow$ can be written as $.[\langle\rightarrow\rangle]/\uparrow/\downarrow[\langle\leftarrow\rangle]$, and likewise for $\leftarrow$. This helps reduce the number of cases.

Step 2. A normal form: a path expression is separated if it is a union of expressions of the form α/β, with α walking upwards and β downwards in the tree.

Step 3. The translation itself from formulas $\phi(x, y) \in FO + TC^1_{np}$ to separated Regular XPath$^{\approx}$ path expressions.
To enable an inductive translation, we use conjunctive tree queries over path expressions.

Crucial lemma: showing that the separated expressions are closed under $*$.

Proof sketch

Difficult direction: $FO + TC^1_{np} \subseteq$ Regular XPath$^{\approx}$

Step 1. Restrict attention to binary branching trees. On such trees $\rightarrow$ can be written as $.[\langle\rightarrow\rangle]/\uparrow/\downarrow[\langle\leftarrow\rangle]$, and likewise for $\leftarrow$. This helps reduce the number of cases.

Step 2. A normal form: a path expression is separated if it is a union of expressions of the form α/β, with α walking upwards and β downwards in the tree.

Step 3. The translation itself from formulas $\phi(x, y) \in FO + TC^1_{np}$ to separated Regular XPath$^{\approx}$ path expressions.
To enable an inductive translation, we use conjunctive tree queries over path expressions.

Crucial lemma: showing that the separated expressions are closed under $*$.

Proof sketch

Difficult direction: $FO + TC^1_{np} \subseteq$ Regular XPath$^{\approx}$

Step 1. Restrict attention to binary branching trees. On such trees $\rightarrow$ can be written as $.[\langle\rightarrow\rangle]/\uparrow/\downarrow[\langle\leftarrow\rangle]$, and likewise for $\leftarrow$. This helps reduce the number of cases.

Step 2. A normal form: a path expression is separated if it is a union of expressions of the form α/β, with α walking upwards and β downwards in the tree.

Step 3. The translation itself from formulas $\phi(x, y) \in FO + TC^1_{np}$ to separated Regular XPath$^{\approx}$ path expressions.
To enable an inductive translation, we use conjunctive tree queries over path expressions.

Crucial lemma: showing that the separated expressions are closed under $*$.

Proof sketch

Difficult direction: $FO + TC^1_{np} \subseteq$ Regular XPath$^{\approx}$

Step 1. Restrict attention to binary branching trees. On such trees $\rightarrow$ can be written as $.[\langle\rightarrow\rangle]/\uparrow/\downarrow[\langle\leftarrow\rangle]$, and likewise for $\leftarrow$. This helps reduce the number of cases.

Step 2. A normal form: a path expression is separated if it is a union of expressions of the form α/β, with α walking upwards and β downwards in the tree.

Step 3. The translation itself from formulas $\phi(x, y) \in FO + TC^1_{np}$ to separated Regular XPath$^{\approx}$ path expressions.
To enable an inductive translation, we use conjunctive tree queries over path expressions.

Crucial lemma: showing that the separated expressions are closed under $*$.

Separated path expressions are closed under $*$

Separated normal form

A path expression is separated if it is of the form $\bigcup_i(\alpha_i/\beta_i)$, with each α_i walking upwards and β_i downwards in the tree (but allowing arbitrary tests).

Example:
How to separate $(\uparrow[p]/\uparrow[q]/\downarrow[r])^*$?

Answer:
$\uparrow[p]/\Big(\uparrow[p \wedge q \wedge \langle\downarrow[r]\rangle]\Big)^*/\uparrow[q]/\downarrow[r] \;\; \cup \;\; .$

The general case: use loop to cut all detours short.

Separated path expressions are closed under $*$

Separated normal form

A path expression is separated if it is of the form $\bigcup_i (\alpha_i/\beta_i)$, with each α_i walking upwards and β_i downwards in the tree (but allowing arbitrary tests).

Example:
How to separate $\left(\uparrow[p]/\uparrow[q]/\downarrow[r]\right)^*$?

Answer:
$\uparrow[p]/\left(\uparrow[p \wedge q \wedge \langle\downarrow[r]\rangle]\right)^*/\uparrow[q]/\downarrow[r] \ \cup \ .$

The general case: use loop to cut all detours short.

Separated path expressions are closed under $*$

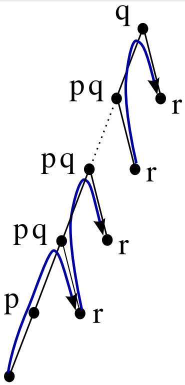
Separated normal form

A path expression is separated if it is of the form $\bigcup_i(\alpha_i/\beta_i)$, with each α_i walking upwards and β_i downwards in the tree (but allowing arbitrary tests).

Example:
How to separate $(\uparrow[p]/\uparrow[q]/\downarrow[r])^*$?

Answer:
$\uparrow[p]/\Big(\uparrow[p \wedge q \wedge \langle\downarrow[r]\rangle]\Big)^*/\uparrow[q]/\downarrow[r] \;\; \cup \;\; .$

The general case: use loop to cut all detours short.



Separated path expressions are closed under $*$

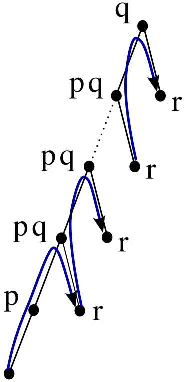
Separated normal form

A path expression is separated if it is of the form $\bigcup_i (\alpha_i/\beta_i)$, with each α_i walking upwards and β_i downwards in the tree (but allowing arbitrary tests).

Example:
How to separate $\left(\uparrow[p]/\uparrow[q]/\downarrow[r]\right)^*$?

Answer:
$\uparrow[p]/\left(\uparrow[p \wedge q \wedge \langle\downarrow[r]\rangle]\right)^*/\uparrow[q]/\downarrow[r] \ \cup \ .$

The general case: use loop to cut all detours short.



Summary of results

$$\begin{array}{clcll}
 & \mu \textit{Regular XPath} & \equiv & & \textit{MSO}[\downarrow, \rightarrow] \\
 & \rotatebox{90}{$\subseteq$} & & & \\
\rotatebox{90}{$\subsetneq$}\ (1) & \textit{Regular XPath}^{\approx} & \equiv & & \textit{FO} + \textit{TC}^1_{np}[\downarrow, \rightarrow] \\
 & \rotatebox{90}{$\subseteq$} & & & \\
 & *\text{-positive } \textit{Regular XPath}^{\approx} & \equiv & & \textit{FO} + \textit{posTC}^1_{np}[\downarrow, \rightarrow] \\
 & \rotatebox{90}{$\subsetneq$} & & & \\
 & \textit{Conditional XPath} & \equiv & (2) & \textit{FO}[\downarrow^*, \rightarrow^*] \\
 & \rotatebox{90}{$\subsetneq$} & & & \\
 & \textit{Core XPath} & \equiv & (3) & \textit{FO}^2[\downarrow, \rightarrow, \downarrow^*, \rightarrow^*]
\end{array}$$

(1) Bojańczyk et al. (2006 ICALP)
(2) Marx (2004 PODS)
(3) Marx & De Rijke (2005 SIGMOD Record), only for absolute path expr's

Questions

- Is Regular XPath strictly contained in Regular XPath$^{\approx}$? (does loop really contribute to the expressive power of Regular XPath$^{\approx}$?)
- Is $FO + (pos)TC^1_{np}$ strictly contained in $FO + (pos)TC^1$?
- Is $FO + TC^1_{np}$ strictly contained in MSO?
- Does Regular XPath or Regular XPath$^{\approx}$ admit an automata theoretic characterization, and, if so, can we use it to answer questions such as the above?
  - Partial result: a characterization for the $*$-positive fragment.

Questions

- Is Regular XPath strictly contained in Regular XPath$^{\approx}$? (does loop really contribute to the expressive power of Regular XPath$^{\approx}$?)
- Is $FO + (pos)TC^1_{np}$ strictly contained in $FO + (pos)TC^1$?
- Is $FO + TC^1_{np}$ strictly contained in MSO?
- Does Regular XPath or Regular XPath$^{\approx}$ admit an automata theoretic characterization, and, if so, can we use it to answer questions such as the above?
  - Partial result: a characterization for the $*$-positive fragment.

Questions

- Is Regular XPath strictly contained in Regular XPath$^{\approx}$? (does loop really contribute to the expressive power of Regular XPath$^{\approx}$?)
- Is $FO + (pos)TC^1_{np}$ strictly contained in $FO + (pos)TC^1$?
- Is $FO + TC^1_{np}$ strictly contained in MSO?
- Does Regular XPath or Regular XPath$^{\approx}$ admit an automata theoretic characterization, and, if so, can we use it to answer questions such as the above?
  - Partial result: a characterization for the $*$-positive fragment.

Questions

- Is Regular XPath strictly contained in Regular XPath$^{\approx}$? (does loop really contribute to the expressive power of Regular XPath$^{\approx}$?)
- Is $FO + (pos)TC^1_{np}$ strictly contained in $FO + (pos)TC^1$?
- Is $FO + TC^1_{np}$ strictly contained in MSO?
- Does Regular XPath or Regular XPath$^{\approx}$ admit an automata theoretic characterization, and, if so, can we use it to answer questions such as the above?
  - Partial result: a characterization for the $*$-positive fragment.

Questions

- Is Regular XPath strictly contained in Regular XPath$^{\approx}$? (does loop really contribute to the expressive power of Regular XPath$^{\approx}$?)
- Is $FO + (pos)TC^1_{np}$ strictly contained in $FO + (pos)TC^1$?
- Is $FO + TC^1_{np}$ strictly contained in *MSO*?
- Does Regular XPath or Regular XPath$^{\approx}$ admit an automata theoretic characterization, and, if so, can we use it to answer questions such as the above?
  - Partial result: a characterization for the $*$-positive fragment.

Pebble tree walking automata

- Let's consider (non-deterministic) pebble tree walking automata with the following types of pebbles:
  - weak pebbles can only be lifted when the automaton is visiting the relevant node.
  - strong pebbles can be lifted from anywhere.
  - non-inspectable weak pebbles are weak pebbles that cannot be inspected. Note: the automaton might not know for sure whether lifting is a valid move! It may crash.
  - return pebbles can be lifted from anywhere, with the side effect that the automaton moves to the relevant node. Furthermore, these pebbles cannot be inspected.
- We use these automata to accept paths: the automata start somewhere in the tree and finish somewhere.

Pebble tree walking automata

- Let's consider (non-deterministic) pebble tree walking automata with the following types of pebbles:
  - weak pebbles can only be lifted when the automaton is visiting the relevant node.
  - strong pebbles can be lifted from anywhere.
  - non-inspectable weak pebbles are weak pebbles that cannot be inspected. Note: the automaton might not know for sure whether lifting is a valid move! It may crash.
  - return pebbles can be lifted from anywhere, with the side effect that the automaton moves to the relevant node. Furthermore, these pebbles cannot be inspected.
- We use these automata to accept paths: the automata start somewhere in the tree and finish somewhere.

Pebble tree walking automata

- Let's consider (non-deterministic) pebble tree walking automata with the following types of pebbles:
  - weak pebbles can only be lifted when the automaton is visiting the relevant node.
  - strong pebbles can be lifted from anywhere.
  - non-inspectable weak pebbles are weak pebbles that cannot be inspected. Note: the automaton might not know for sure whether lifting is a valid move! It may crash.
  - return pebbles can be lifted from anywhere, with the side effect that the automaton moves to the relevant node. Furthermore, these pebbles cannot be inspected.
- We use these automata to accept paths: the automata start somewhere in the tree and finish somewhere.

Pebble tree walking automata

- Let's consider (non-deterministic) pebble tree walking automata with the following types of pebbles:
  - weak pebbles can only be lifted when the automaton is visiting the relevant node.
  - strong pebbles can be lifted from anywhere.
  - non-inspectable weak pebbles are weak pebbles that cannot be inspected. Note: the automaton might not know for sure whether lifting is a valid move! It may crash.
  - return pebbles can be lifted from anywhere, with the side effect that the automaton moves to the relevant node. Furthermore, these pebbles cannot be inspected.
- We use these automata to accept paths: the automata start somewhere in the tree and finish somewhere.

Pebble tree walking automata

- Let's consider (non-deterministic) pebble tree walking automata with the following types of pebbles:
  - weak pebbles can only be lifted when the automaton is visiting the relevant node.
  - strong pebbles can be lifted from anywhere.
  - non-inspectable weak pebbles are weak pebbles that cannot be inspected. Note: the automaton might not know for sure whether lifting is a valid move! It may crash.
  - return pebbles can be lifted from anywhere, with the side effect that the automaton moves to the relevant node. Furthermore, these pebbles cannot be inspected.
- We use these automata to accept paths: the automata start somewhere in the tree and finish somewhere.

Pebble tree walking automata

- Let's consider (non-deterministic) pebble tree walking automata with the following types of pebbles:
  - weak pebbles can only be lifted when the automaton is visiting the relevant node.
  - strong pebbles can be lifted from anywhere.
  - non-inspectable weak pebbles are weak pebbles that cannot be inspected. Note: the automaton might not know for sure whether lifting is a valid move! It may crash.
  - return pebbles can be lifted from anywhere, with the side effect that the automaton moves to the relevant node. Furthermore, these pebbles cannot be inspected.
- We use these automata to accept paths: the automata start somewhere in the tree and finish somewhere.

Some partial results

- **Thm:** $*$-positive Regular XPath$^{\approx}$ (hence also FO+posTC$^1_{np}$) can define the same binary relations as twa with non-inspectable weak pebbles.
  (extends a result of Goris and Marx '05)

- **Thm (Engelfriet and Hoogeboom '05):** $*$-positive FO+TC^1 can define the same binary relations as twa with strong pebbles.
- **Thm (Bojanczyk e.a. '06):** Weak pebbles suffice.

Call an Regular XPath expression positive if it uses only atomic negation, of the from $\neg\langle\uparrow\rangle$, $\neg\langle\downarrow\rangle$, $\neg\langle\leftarrow\rangle$ and $\neg\langle\rightarrow\rangle$

- **Thm:** Positive Regular XPath can define the same binary relations as twa with return pebbles.

Some partial results

- **Thm:**
  $*$-positive Regular XPath$^{\approx}$ (hence also FO+posTC$^1_{np}$) can define the same binary relations as twa with non-inspectable weak pebbles.
  (extends a result of Goris and Marx '05)

- **Thm (Engelfriet and Hoogeboom '05):** $*$-positive FO+TC^1 can define the same binary relations as twa with strong pebbles.
- **Thm (Bojanczyk e.a. '06):** Weak pebbles suffice.

Call an Regular XPath expression positive if it uses only atomic negation, of the from $\neg\langle\uparrow\rangle$, $\neg\langle\downarrow\rangle$, $\neg\langle\leftarrow\rangle$ and $\neg\langle\rightarrow\rangle$

- **Thm:** Positive Regular XPath can define the same binary relations as twa with return pebbles.

Some partial results

- **Thm:**
  $*$-positive Regular XPath$^{\approx}$ (hence also FO+posTC$^1_{np}$) can define the same binary relations as twa with non-inspectable weak pebbles.
  (extends a result of Goris and Marx '05)

- **Thm (Engelfriet and Hoogeboom '05):** $*$-positive FO+TC^1 can define the same binary relations as twa with strong pebbles.
- **Thm (Bojanczyk e.a. '06):** Weak pebbles suffice.

Call an Regular XPath expression positive if it uses only atomic negation, of the from $\neg\langle\uparrow\rangle$, $\neg\langle\downarrow\rangle$, $\neg\langle\leftarrow\rangle$ and $\neg\langle\rightarrow\rangle$

- **Thm:** Positive Regular XPath can define the same binary relations as twa with return pebbles.

Some partial results

- **Thm:**
  $*$-positive Regular XPath$^{\approx}$ (hence also FO+posTC$^1_{np}$) can define the same binary relations as twa with non-inspectable weak pebbles.
  (extends a result of Goris and Marx '05)

- **Thm (Engelfriet and Hoogeboom '05):** $*$-positive FO+TC^1 can define the same binary relations as twa with strong pebbles.
- **Thm (Bojanczyk e.a. '06):** Weak pebbles suffice.

Call an Regular XPath expression positive if it uses only atomic negation, of the from $\neg\langle\uparrow\rangle$, $\neg\langle\downarrow\rangle$, $\neg\langle\leftarrow\rangle$ and $\neg\langle\rightarrow\rangle$

- **Thm:** Positive Regular XPath can define the same binary relations as twa with return pebbles.

- **Thm:**
  $*$-positive Regular XPath$^{\approx}$ (hence also FO+posTC$^1_{np}$) can define the same binary relations as twa with non-inspectable weak pebbles.
  (extends a result of Goris and Marx '05)

- **Thm (Engelfriet and Hoogeboom '05):** $*$-positive FO+TC^1 can define the same binary relations as twa with strong pebbles.
- **Thm (Bojanczyk e.a. '06):** Weak pebbles suffice.

Call an Regular XPath expression positive if it uses only atomic negation, of the from $\neg\langle\uparrow\rangle$, $\neg\langle\downarrow\rangle$, $\neg\langle\leftarrow\rangle$ and $\neg\langle\rightarrow\rangle$

- **Thm:** Positive Regular XPath can define the same binary relations as twa with return pebbles.

Some partial results

- **Thm:**
  $*$-positive Regular XPath$^{\approx}$ (hence also FO+posTC$^1_{np}$) can define the same binary relations as twa with non-inspectable weak pebbles.
  (extends a result of Goris and Marx ’05)

- **Thm (Engelfriet and Hoogeboom ’05):** $*$-positive FO+TC^1 can define the same binary relations as twa with strong pebbles.
- **Thm (Bojanczyk e.a. ’06):** Weak pebbles suffice.

Call an Regular XPath expression positive if it uses only atomic negation, of the from $\neg\langle\uparrow\rangle$, $\neg\langle\downarrow\rangle$, $\neg\langle\leftarrow\rangle$ and $\neg\langle\rightarrow\rangle$

- **Thm:** Positive Regular XPath can define the same binary relations as twa with return pebbles.

That's all.