DETERMINISTIC CATERPILLAR EXPRESSIONS

Kai Salomaa[(2)] *Sheng Yu*[(1)] *Jinfeng Zan*[(2)]

[(1)]Department of Computer Science, Univesity of Western Ontario, London, Ontario, Canada

[(2)]School of Computing, Queen's University, Kingston, Ontario, Canada

Background

- *Caterpillar expressions* (Brüggemann-Klein & Wood, 2000) have been used for the specification of context in structured documents.
- A caterpillar expression is a regular expression built from *atomic instructions* that defines the computation of a tree walking automaton.
- An atomic instruction consists of a move to an adjacent node or a test instruction (a test on the label of the current node, or e.g. a test whether the node is a leaf).

- A caterpillar expression defines the operation of a tree walking automaton, or sequential tree automaton:
  - at a given time the finite state control is located at one node of the tree.
- “Parallel” bottom-up or top-down tree automata define the *regular tree languages.*
- Bojańczyk and Colcombet (STOC 2005) have shown that (sequential) tree walking automata cannot recognize all regular tree languages.

- Classical applications of tree automata use trees over *ranked alphabets:* each node label has a rank that determines the number of children.

- The new applications (like the manipulation of XML documents) use *trees over unranked alphabets:* there is no apriori upper bound for the number of children of a node (but the number of children is finite).

- In the unranked case, a finite transition function cannot directly specify an arbitrary child node where the automaton moves to: *Caterpillar expressions* provide a convenient mechanism to specify the operation of the tree walking automaton.

Definition. Let Σ be the (unranked) alphabet used to label nodes of the trees.

The set of *atomic caterpillar expressions* is

$$\begin{aligned}\Delta \quad = \quad & \Sigma \cup \{isFirst, isLast, isLeaf, isRoot, \\ & Up, Left, Right, First, Last\}.\end{aligned}$$

- An instruction $a \in \Sigma$ tests whether the label of the current node is a.
- The instructions $isFirst$, $isLast$, $isLeaf$ and $isRoot$ test whether the current node is the first (leftmost) sibling of its parent, the last sibling, a leaf node or the root node.

- The *move instructions* Up, $Left$, $Right$, $First$ and $Last$, respectively, instruct the caterpillar to move from the current node to its parent, the next sibling to the left, the next sibling to the right, the leftmost child of the current node, or the rightmost child of the current node.

- A *caterpillar expression* is a regular expression α over Δ.

- Computations of α are defined by all sequences of instructions in the language denoted by α, $L(\alpha)$.
  (The formal definition of the computations is given in the proceedings.)

Example. Let $\Sigma = \{a, b\}$.

$$\begin{aligned} \alpha_1 \quad = \quad & (First \cdot Right^*)^* \cdot isFirst \cdot (isLeaf \cdot a \cdot Right) \cdot \\ & (isLeaf \cdot b \cdot Right) \cdot (isLeaf \cdot a \cdot isLast) \end{aligned}$$

- α_1 defines the set of trees that contain a node with precisely three children that are all leaves and labeled, respectively, by a, b, a.
- The subexpression $(First \cdot Right^*)^*$ moves the caterpillar (nondeterministically) to an arbitrary node of the tree.
- The remaining part of α_1 checks the required property.

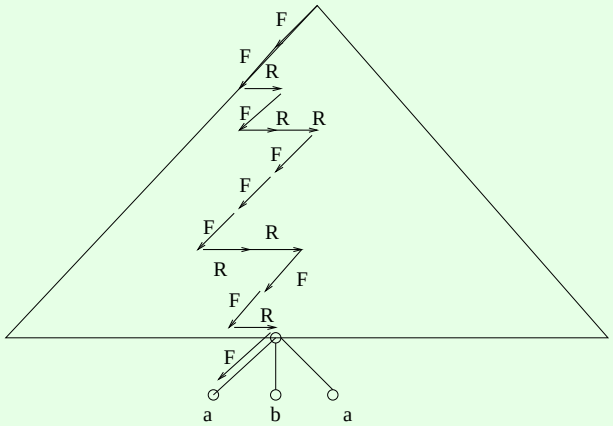

Figure 1: Example continued: Instructions defined by $(First \cdot Right^*)^*$ find an arbitrary first child in the tree. Here “F” denotes “First”, “R” denotes “Right”.

The computation of the caterpillar α_1 uses *nondeterminism* to find an arbitrary node of the tree.

Theorem 1. *Caterpillar expressions define the same family of tree languages as nondeterministic tree walking automata.*

Proof idea. Given a tree walking automaton A, we can define a caterpillar expression α_A such that an instruction sequence encoded by α_A can be successfully executed on a tree t *iff* it corresponds to a valid computation of A on t, with state information deleted.
The proof can be found in the proceedings.

- Nondeterministic tree walking automata are strictly more powerful than deterministic tree walking automata (Bojańczyk & Colcombet, ICALP 2004)

- *Crucial question:* Is the computation defined by a given caterpillar expression deterministic?

- In their original paper, Brüggemann-Klein & Wood considered only the "operational" definition of determinism: a caterpillar is deterministic if the tree walking automaton simulating the computation is deterministic.

- In order to algorithmically decide determinism of given caterpillar expressions, we need a more direct definition of determinism:
  - determinism is defined in terms of the sequences of instructions denoted by an expression.

  Intuitively, we say that two distinct atomic instructions $c, c' \in \Delta$ are *mutually exclusive* if at any node of an arbitrary tree at most one of the instructions c and c' can be successfully executed.

Definition. Let $c, c' \in \Delta$.

We say that instructions c and c' are *mutually exclusive* if either

(i) $c, c' \in \Sigma$ and $c \neq c'$, that is, c and c' are tests on distinct symbols of Σ, or,

(ii) $\{c, c'\}$ is one of the sets $\{First, isLeaf\}$, $\{Last, isLeaf\}$, $\{Up, isRoot\}$, $\{Left, isFirst\}$, or $\{Right, isLast\}$.

The above are exactly all pairs of instructions that cannot both be successfully executed at any node.

Definition of determinism:

A caterpillar expression α is *deterministic* if:

Whenever wc_1w_1 and wc_2w_2 are in $L(\alpha)$ where $w, w_1, w_2 \in \Delta^*$, $c_1, c_2 \in \Delta$, $c_1 \neq c_2$, then c_1 and c_2 are mutually exclusive.

Interpretation: for any instruction sequences w and w' defined by α that are not prefixes of one another, the pair of instructions following the longest common prefix of w and w' has to be mutually exclusive.

Example

Define

$$\begin{aligned}\alpha_{\mathrm{trav}} \quad = \quad & \mathrm{First}^* \cdot \mathrm{isLeaf} \cdot (\mathrm{Right} \cdot \mathrm{First}^* \cdot \mathrm{isLeaf})^* \cdot \mathrm{isLast} \cdot \\ & (\mathrm{Up} \cdot (\mathrm{Right} \cdot \mathrm{First}^* \cdot \mathrm{isLeaf})^* \cdot \mathrm{isLast})^* \cdot \mathrm{isRoot}\end{aligned}$$

- α_{trav} is deterministic: possible choices for the next instruction are $\{\mathrm{First}, \mathrm{isLeaf}\}$, $\{\mathrm{Right}, \mathrm{isLast}\}$ and $\{\mathrm{Up}, \mathrm{isRoot}\}$ and these are all mutually exclusive.

- The subexpression $\mathrm{First}^* \cdot \mathrm{isLeaf}$ finds the leftmost leaf of the tree.
- The next subexpression $(\mathrm{Right} \cdot \mathrm{First}^* \cdot \mathrm{isLeaf})^* \cdot \mathrm{isLast}$ finds the leftmost leaf of the current subtree that is the last child of its parent.
- The process is then iterated by going one step up in the subexpression $(\mathrm{Up} \cdot \ldots \cdot \mathrm{isLast})^*$.
- *We can verify:* The instruction sequences of α_{trav} force the caterpillar to traverse an arbitrary input tree in depth-first left-to-right order.

Question: Given a caterpillar expression α, how can we test whether or not α is deterministic?

Can we use some existing algorithms?

- (Possibly) related notions for regular expressions include *unambiguity* and *one-unambiguity.*

Unambiguity:

- Mark different occurrences of the same symbol with subscripts. For example, expression $(a+b)^*aa^*$ becomes $(a_1+b_1)^*a_2a_3^*$.
- Now the word baa has two witnesses: $b_1a_1a_2$ and $b_1a_2a_3$.
- A regular expression is unambiguous if no word has more than one witness.
- Thus, $(a+b)^*aa^*$ is ambiguous. The same language is denoted by the unambiguous regular expression $(a+b)^*a$.

One-ambiguity:

- We additionally require that the choice for the match of the next symbol has to be made without looking at the following symbols in the word.

There exist efficient algorithms to test for

- unambiguity: Book et al., 1971,
- one-unambiguity: Brüggemann-Klein & Wood, 1998.

However: it turns out that deterministic caterpillar expressions need not be (one-)unambiguous or vice versa.

- For example, the expression $(a + Up)^* First$ is one-unambiguous but it is *not* deterministic.
  (Other cases verified similarly.)

In order to test for determinism, we need to look for other approaches ...

We develop two polynomial time algorithms to test determinism of caterpillar expressions:

- **Algorithm 1** relies only on structural properties of the expression and works for a restricted class of caterpillar expressions (that define instruction languages having polynomial density).

- **Algorithm 2** relies on techniques that have been previously used to test code properties of regular languages. `Algorithm 2` places no restrictions on the input expressions.

Recall *(Szilard, Yu, Zhang, Shallit 1992)*:

Proposition. A regular language R over Δ has density in $O(n^k)$, $k \geq 0$, iff R can be denoted by a finite union of regular expressions of the form

$$w_0 u_1^* w_1 u_2^* \dots u_{m+1}^* w_{m+1}, \quad m \leq k \tag{1}$$

where $w_i, u_j \in \Delta^*$, $i = 0, \dots m+1$, $j = 1, \dots m+1$.

We call finite unions of regular expressions as in (1), k*-bounded regular expressions.*

Algorithm 1:

- The input expressions are restricted to be k-bounded (with fixed k).
- The input expression is transformed to a sum of *normalized expressions* for which determinism can be decided easily.

- We consider expressions of the form

$$x_0 y_1^* x_1 y_2^* x_2 \cdots y_{m+1}^* x_{m+1}, \quad x_i, y_j \in \Delta^*,\ m \leq k. \tag{2}$$

- Without loss of generality $y_j \neq \lambda$, j = 1, . . . , m+1, (possible empty y_j's can be omitted).

Definition. An expression (2) is *normalized* if

1. for each $1 \leq i \leq m+1$, longest common prefix of x_i and y_i is λ, and

2. $x_i \neq \lambda$, for each $1 \leq i \leq m$.

Lemma 1. *An arbitrary k-bounded expression*

$$\alpha = x_0 y_1^* x_1 y_2^* x_2 \cdots y_{m+1}^* x_{m+1},$$

$m \leq k$, can be written as the sum of $O(n^k)$ normalized expressions each having length $O(k \cdot n)$.
Here n is the length of α.

The proof of Lemma 1 is included in the full version of the paper. The proof uses techniques from combinatorics on words (the Lyndon-Schützenberger theorem).

Definition. An expression $\alpha = x_0 y_1^* x_1 y_2^* x_2 \cdots y_{m+1}^* x_{m+1}$ is *well-behaved* if $x_i \neq \lambda$ implies that $\mathrm{first}(y_i)$ and $\mathrm{first}(x_i)$ are mutually exclusive, $1 \leq i \leq m+1$.

Lemma 2. *If α (as above) is normalized and deterministic, then α is well-behaved.*

Proof: After executing y_i, the next instruction is $\mathrm{first}(y_i)$ or $\mathrm{first}(x_i)$. Since α is normalized, these are distinct. Hence they have to be mutually exclusive. □

Algorithm 1:

- Due to Lemmas 1 and 2, in order to test determinism of k-bounded expressions, it is sufficient to consider sums of well-behaved normalized expressions.

- *Rough idea:* The algorithm looks for prefixes of two well-behaved normalized expressions where the next symbols would not be mutually exclusive. Since the expressions contain at most k consecutive stars, where k is fixed, the running time can be bounded by a polynomial.
  The details of the algorithm are given in the proceedings.

- The previous algorithm assumes that the input expressions define an instruction language of polynomial density.

- It would not be difficult to decide determinism of a caterpillar expression α if we could construct a DFA for the instruction language of α (. . . this requires exponential time).

- The general algorithm (**Algorithm 2**) constructs an NFA A_α equivalent with α and then tests a property of the state-pair graph of A_α,
  (or the *square* of A_α (Berstel & Perrin, 1985)).

Let $A = (\Omega, Q, q_0, F, \delta)$ be an NFA.

Definition. The *state-pair graph* of A is the directed graph $G_A = (V, E)$ where the set of nodes is $V = Q \times Q$ and the set of Ω-labeled edges is

$$E = \{((p,q), b, (p',q')) \mid (p,b,p') \in \delta, (q,b,q') \in \delta, \quad b \in \Omega\}.$$

Intuitively, the edges connect pairs of states of A that can be reached from each other on a transition with the same symbol.

Note: our definition of determinism refers to the instruction language ($\subseteq \Delta^*$) defined by a caterpillar expression, i.e.,

- determinism is a property of subsets of Δ^*.

Lemma 3. *Assume $A = (\Delta, Q, q_0, F, \delta)$ is a reduced NFA (where Δ is a caterpillar alphabet).*

The language $L(A)$ is <u>not</u> deterministic iff there exist $p, q \in Q$ such that

(i) *The state-pair graph G_A has a path from (q_0, q_0) to (p, q).*

(ii) *There exist $c_1, c_2 \in \Delta$, $c_1 \neq c_2$, such that $(p, c_1, p') \in \delta$ and $(q, c_2, q') \in \delta$ for some $p', q' \in Q$, and c_1, c_2 are not mutually exclusive.*

- The proof of Lemma 3 is given in the proceedings.
- Given a caterpillar expression α we can construct in time $O(n^2 \log^4 n)$ the state-pair graph of an NFA that recognizes the instruction language $L(\alpha)$.
- Hence using a standard graph reachability algorithm we have the following:

Theorem 2. *Given a caterpillar expression α over Δ we can decide in polynomial time whether or not α is deterministic.*

- In Theorem 1, we have shown that general caterpillar expressions can simulate arbitrary nondeterministic tree walking automata. (The converse simulation is obvious.)

- When applying the construction to a deterministic tree walking automaton the resulting caterpillar is, in general, not deterministic.

Open problem: Do the deterministic tree walking automata define a strictly larger family of tree languages than the deterministic caterpillar expressions?