Annotated Stack Trees*

M. Hague[1] and V. Penelle[2]

[1] Royal Holloway, University of London
[2] LIGM, Université Paris-Est

Abstract

Annotated pushdown automata provide an automaton model of higher-order recursion schemes, which may in turn be used to model higher-order programs for the purposes of verification. We study Ground Annotated Stack Tree Rewrite Systems – a tree rewrite system where each node is labelled by the configuration of an annotated pushdown automaton. This allows the modelling of fork and join constructs in higher-order programs and is a generalisation of higher-order stack trees recently introduced by Penelle.

We show that, given a regular set of annotated stack trees, the set of trees that can reach this set is also regular, and constructible in n-EXPTIME for an order-n system, which is optimal. We also show that our construction can be extended to allow a global state through which unrelated nodes of the tree may communicate, provided the number of communications is subject to a fixed bound.

1 Introduction

Modern day programming increasingly embraces higher-order programming, both via the inclusion of higher-order constructs in languages such as C++, JavaScript and Python, but also via the importance of *callbacks* in highly popular technologies such as jQuery and Node.js. For example, to read a file in Node.js, one would write

```
fs.readFile('f.txt', function (err, data) { ..use data.. });
```

In this code, the call to `readFile` spawns a new thread that asynchronously reads `f.txt` and sends the `data` to the function argument. This function will have access to, and frequently use, the closure information of the scope in which it appears. The rest of the program runs *in parallel* with this call. This style of programming is fundamental to both jQuery and Node.js programming, as well as being a popular for programs handling input events or slow IO operations such as fetching remote data or querying databases (e.g. HTML5's indexedDB).

Analysing such programs is a challenge for verification tools which usually do not model higher-order recursion, or closures, accurately. However, several higher-order model-checking tools have been recently developed. This trend was pioneered by Kobayashi *et al.* [17] who developed an *intersection type* technique for analysing *higher-order recursion schemes* – a model of higher-order computation. This was implemented in the TRecS tool [16] which demonstrated the feasibility of higher-order model-checking in practice, despite the high theoretical complexities ($(n-1)$-EXPTIME for an order-n recursion scheme). This success has led to the development of several new tools for analysing recursion schemes: GTRecS [18, 20], TravMC [27], C-SHORe [6], HorSat [7], and Preface [31].

In particular, the C-SHORe tool is based on an automata model of recursion schemes called *annotated (or collapsible) pushdown systems* [14]. This is a generalisation of pushdown systems – which accurately model first-order recursion – to the higher-order case. C-SHORe implements

*This work was supported by the Engineering and Physical Sciences Research Council [EP/K009907/1].

a *saturation* algorithm to perform a backwards reachability analysis, which first appeared in ICALP 2012 [5]. Saturation was popularised by Bouajjani *et al.* [1] for the analysis of pushdown systems, which was implemented in the successful Moped tool [34, 36].

Contributions In this work we introduce a generalisation of annotated pushdown systems: *ground annotated stack tree rewrite systems (GASTRS)*. A configuration of a GASTRS is an *annotated stack tree* – that is, a tree where each node is labelled by the configuration of an annotated pushdown system. Operations may update the leaf nodes of the tree, either by updating the configuration, creating new leaf nodes, or destroying them. Nodes are created and destroyed using

$$p \xrightarrow{+} (p_1, \ldots, p_m) \text{ and } (p'_1, \ldots, p'_m) \xrightarrow{-} p'$$

which can be seen as spawning m copies of the current process (including closure information) using the first rule, and then later joining these processes with the second rule, returning control to the previous execution (parent node). Alternatively, we can just use $p \xrightarrow{+} (p_1, p_2)$ for a basic fork that does not join.

This model is a generalisation of *higher-order stack trees* recently introduced by Penelle [30], where the tree nodes are labelled by a restriction of annotated pushdown automata called *higher-order pushdown automata*.

As our main contribution, we show that the global backwards reachability problem for GASTRSs can be solved via a saturation technique. That is, given a regular target set of annotated stack trees, we compute a regular representation of all trees from which there is a run of the system to the target set. Note that being able to specify a target set of trees allows us to identify error states such as race conditions between threads. Our result is a generalisation of the ICALP 2012 algorithm, and as such, may be implemented as part of the C-SHORe tool.

Moreover, we define a notion of regularity amenable to saturation which is also closed under the standard boolean operations.

As a final contribution, we show that the model can be extended to allow a bounded amount of communication between separate nodes of the tree. I.e., we add a global state to the system and perform a "context-bounded" analysis [32], where the global state can only be changed an *a priori* fixed number of times.

Related Work Annotated pushdown systems are a generalisation of higher-order pushdown systems that provide a model of recursion schemes subject to a technical constraint called *safety* [26, 15] and are closely related to the Caucal hierarchy [9]. Parys has shown that safety is a genuine constraint on definable traces [29]. Panic automata provided the first model of order-2 schemes, while annotated pushdown systems model schemes of arbitrary order. These formalisms have good model-checking properties. E.g. μ-calculus decidability [28, 14]. Krivine machines can also be used to model recursion schemes [33].

There has been some work studying concurrent variants of recursion scheme model checking, including a context-bounded algorithm for recursion schemes [19], and further underapproximation methods such as phase-bounded, ordered, and scope-bounding [12, 35]. These works allow only a fixed number of threads.

Dynamic thread creation is permitted by both Yasukata *et al.* [37] and by Chadha and Viswanathan [10]. In Yasukata *et al.*'s model, recursion schemes may spawn and join threads. Communication is permitted only via nested locks, whereas in our model we allow shared memory, but only a bounded number of memory updates. Their work is a generalisation of results for order-1 pushdown systems [11]. Chadha and Viswanathan allow threads to be spawned, but only one thread runs at a time, and must run to completion. Moreover, the tree structure is not maintained.

Saturation methods also exist for *ground tree rewrite systems* and related systems [24, 4, 25], though use different techniques. Our context-bounded model relates to weak GTRS with state introduced by Lin [23]. Adding such weak state to process rewrite systems was considered by Kretínský *et al.* [21].

A saturation technique has also been developed for dynamic trees of pushdown processes [3]. These are trees where each process on each node is active (in our model, only the leaf nodes are active). However, their spawn operations do not copy the current process, losing closure information. It would be interesting and non-trivial to study the combination of both approaches.

Penelle proves decidability of first order logic with reachabilty over rewriting graphs of ground stack tree rewriting systems [30]. This may be used for a context-bounded reachability result for higher-order stack trees. This result relies on MSO decidability over the configuration graphs of higher-order pushdown automata, through a finite set interpretation of any rewriting graph of a ground stack tree rewriting system into a configuration graph of a higher pushdown automaton. This does not hold for annotated pushdown automata.

2 Preliminaries

Trees

An ordered tree over arity at most d over a set of labels Γ is a tuple $(\mathcal{D}, \lambda)$ where $\mathcal{D} \subset \{1, \ldots, d\}^*$ is a tree domain such that $vi \in \mathcal{D}$ implies $v \in \mathcal{D}$ (prefix closed), and $vj \in \mathcal{D}$ for all $j < i$ (younger-sibling closed), and $\lambda : \mathcal{D} \to \Gamma$ is a labelling of the nodes of the tree. Let $v \preceq v'$ denote that v is an ancestor (inclusive) of v' in the tree. We write $t[v \to \gamma]$ to denote the tree $t' = (\mathcal{D} \cup \{v\}, \lambda')$ where $\lambda'(v) = \gamma$ and $\lambda'(v') = \lambda(v')$ for $v' \neq v$, whenever $t = (\mathcal{D}, \lambda)$ and $\mathcal{D} \cup \{v\}$ is a valid tree domain. We will also write $t' = t \setminus V$ to denote the tree obtained by removing all subtrees rooted at $v \in V$ from t. That is $t' = (\mathcal{D}', \lambda')$ when $t = (\mathcal{D}, \lambda)$ and

$$\begin{array}{rcl} \mathcal{D}' & = & \mathcal{D} \setminus \{v' \mid v \in V \wedge v \preceq v'\} \\ \lambda'(v) & = & \begin{cases} \lambda(v) & v \in \mathcal{D}' \\ \text{undefined} & \text{otherwise.} \end{cases} \end{array}$$

Annotated stacks

Let Σ be a set of stack symbols. An annotated stack of order-n is an order-n stack in which stack symbols are annotated with stacks of order at most n. For the rest of the paper, we fix the maximal order to n, and use k to range between n and 1. We simultaneously define for all $1 \leq k \leq n$, the set $\mathcal{S}^{\Sigma}_{k,n}$ of stacks of order-k whose symbols are annotated by stacks of order at most n. Note, we use subscripts to indicate the order of a stack. We ensure all stacks are finite by using the least fixed-point. When the maximal order n is clear, we write $\mathcal{S}^{\Sigma}_{k}$ instead of $\mathcal{S}^{\Sigma}_{k,n}$.

Definition 2.1 (Annotated Stacks)**.** *The family of sets $\left(\mathcal{S}^{\Sigma}_{k,n}\right)_{1 \leq k \leq n}$ is the smallest family (for point-wise inclusion) such that:*

- *for all $2 \leq k \leq n$, $\mathcal{S}^{\Sigma}_{k,n}$ is the set of all (possibly empty) sequences $[s_1 \ldots s_m]_k$ with $s_1, \ldots, s_m \in \mathcal{S}^{\Sigma}_{k-1,n}$.*

- $\mathcal{S}^{\Sigma}_{1,n}$ *is all sequences* $[a_1^{s_1} \dots a_m^{s_m}]_1$ *with* $m \geq 0$ *and for all* $1 \leq i \leq m$*, j* a_i *is a stack symbol in* Σ *and* s_i *is an annotated stack in* $\bigcup\limits_{1 \leq k \leq n} \mathcal{S}^{\Sigma}_{k,n}$.

We write $s :_k s'$ — where s is order-$(k-1)$ — to denote the stack obtained by placing s on top of s'. That is,

- if $s' = [s_1 \dots s_m]_k$ then $s :_k s' = [s s_1 \dots s_m]_k$, and
- if $s' = [s_1 \dots s_m]_{k'}$ with $k' > k$ then $s :_k s' = [(s :_k s_1)\, s_2 \dots s_m]_{k'}$.

This composition associates to the right. For example, the order-3 stack $\left[[[a^s b]_1]_2\right]_3$ can be written $s_1 :_3 s_2$ where s_1 is the order-2 stack $[[a^s b]_1]_2$ and s_2 is the empty order-3 stack $[]_3$. Then $s_1 :_3 s_1 :_3 s_2$ is $\left[[[a^s b]_1]_2\, [[a^s b]_1]_2\right]_3$.

Note that we cannot write $(s_1 :_k s_2) :_k s_3$ since $(s_1 :_k s_2)$ is not order-$(k-1)$.

Operations on Order-n Annotated Stacks

For a given alphabet Σ, we define the set Ops^{Σ}_n of stack operations inductively as follows:

$$\mathrm{Ops}^{\Sigma}_0 = \{\mathrm{rew}_{a \to b} \mid a, b \in \Sigma\} \qquad \mathrm{Ops}^{\Sigma}_1 = \left\{\mathrm{push}^1_1, \mathrm{pop}_1\right\} \cup \mathrm{Ops}^{\Sigma}_0$$
$$\mathrm{Ops}^{\Sigma}_n = \{\mathrm{push}^n_1, \mathrm{push}_n, \mathrm{pop}_n, \mathrm{collapse}_n\} \cup \mathrm{Ops}^{\Sigma}_{(n-1)}$$

We define each operation for a stack s. Annotations are created by push^k_1, which adds a character to the top of a stack $s :_{(k+1)} s'$ annotated by $\mathrm{pop}_k(s)$. This gives the new character access to the context in which it was created.

1. We set $\mathrm{rew}_{a \to b}\left(a^{s'} :_1 s\right) = b^{s'} :_1 s$.
2. We set $\mathrm{push}^k_1(s) = a^{s_k} :_1 s$ when $s = a^{s_1} :_1 s_2 :_2 \cdots :_k s_k :_{(k+1)} \cdots :_n s_n$.
3. We set $\mathrm{push}_k(s :_k s') = s :_k s :_k s'$.
4. We set $\mathrm{pop}_k(s :_k s') = s'$.
5. We set $\mathrm{collapse}_k\left(a^s :_1 s_1 :_{(k+1)} s_2\right) = s :_{(k+1)} s_2$ when s is order-k and $n > k \geq 1$; and $\mathrm{collapse}_n(a^s :_1 s') = s$ when s is order-n.

3 Annotated Stack Trees

An *annotated stack tree* is a tree whose nodes are labelled by annotated stacks. Furthermore, each leaf node is also labelled with a control state. Let $\mathrm{STrees}^n_{\Sigma}$ denote the set of order-n annotated stack trees over Σ.

Definition 3.1 (Order-n Annotated Stack Trees)**.** *An* order-n annotated stack tree *over an alphabet* Σ *and set of control states* $\mathbb{P}$ *is a* $\left(\mathcal{S}^{\Sigma}_n \cup \left(\mathbb{P} \times \mathcal{S}^{\Sigma}_n\right)\right)$*-labelled tree* $t = (\mathcal{D}, \lambda)$ *such that for all leaves* v *of* t *we have* $\lambda(v) \in \mathbb{P} \times \mathcal{S}^{\Sigma}_n$ *and for all internal nodes* v *of* t *we have* $\lambda(v) \in \mathcal{S}^{\Sigma}_n$.

3.1 Annotated Stack Tree Operations

Definition 3.2 (Order-n Annotated Stack Tree Operations)**.** *Over a given finite alphabet* Σ *and finite set of control states* $\mathbb{P}$*, the set of* order-n stack tree operations *is defined to be*

$$\begin{array}{rcl} \mathrm{STOps}_n^{\Sigma,\mathbb{P}} & = & \left\{ p \xrightarrow{+} (p_1,\ldots,p_m), (p_1,\ldots,p_m) \xrightarrow{-} p \;\mid p, p_1, \ldots, p_m \in \mathbb{P} \right\} \cup \\ & & \left\{ p \xrightarrow{\sigma} p' \;\mid \sigma \in \mathrm{Ops}_n^{\Sigma} \wedge p, p' \in \mathbb{P} \right\} . \end{array}$$

Stack operations may be applied to any leaf of the tree. Let $t_{\bullet_i}$ denote the ith leaf of tree t. We define the local application of a operation to the ith leaf as follows. Let $t = (\mathcal{D}, \lambda)$ and $\lambda(t_{\bullet_i}) = (p, s)$

$$\begin{array}{rcl} \mathrm{Ap}\Big(p \xrightarrow{\sigma} p', i, t\Big) & = & t[t_{\bullet_i} \to (p', \sigma(s))] \\ \mathrm{Ap}\Big(p \xrightarrow{+} (p_1,\ldots,p_m), i, t\Big) & = & t[t_{\bullet_i} \to s][t_{\bullet_i}1 \to (p_1, s)] \cdots [t_{\bullet_i}m \to (p_m, s)] \end{array}$$

and when $t_{\bullet_i} = v1, \ldots, t_{\bullet_{i+m-1}} = vm$ are the only children of v, $\lambda(t_{\bullet_i}) = (p_1, s_1), \ldots, \lambda\big(t_{\bullet_{i+m-1}}\big) = (p_m, s_m)$, and $\lambda(v) = s$,

$$\mathrm{Ap}\Big((p_1,\ldots,p_m) \xrightarrow{-} p, i, t\Big) = \big(t \setminus \big\{t_{\bullet_i}, \ldots, t_{\bullet_{i+m-1}}\big\}\big)[v \to (p, s)] .$$

For all $\theta \in \mathrm{STOps}_n^{\Sigma,\mathbb{P}}$ we write $\theta(t)$ to denote the set $\{t' \mid \exists i. t' = \mathrm{Ap}(\theta, i, t)\}$.

3.2 Ground Annotated Stack Tree Rewrite Systems

Definition 3.3 (Order-n Ground Annotatee Stack Tree Rewrite Systems)**.** *An* order-n ground annotated stack tree rewrite system (GASTRS) $\mathcal{G}$ *is a tuple* $(\Sigma, \mathbb{P}, \mathcal{R})$ *where* Σ *is a finite stack alphabet,* $\mathbb{P}$ *is a finite set of control states, and* $\mathcal{R} \subset \mathrm{STOps}_n^{\Sigma,\mathbb{P}}$ *is a finite set of operations.*

A configuration of an order-n GASTRS is an order-n annotated stack tree t over alphabet Σ. We have a transition $t \to t'$ whenever there is some $\theta \in \mathcal{R}$ and $t' \in \theta(t)$. We write $t \to^* t'$ when there is a run $t = t_0 \to \cdots \to t_m = t'$.

3.3 Regular Sets of Annotated Stack Trees

We define a notion of annotated stack tree automata for recognising regular sets of annotated stack trees. We give an initial exposition here, with more details (definitions and proofs) in Appendix A. In particular, we have the following result.

Proposition 3.1. *Annotated stack tree automata form an effective boolean algebra, membership is in linear time, and emptiness is PSPACE-complete.*

Transitions of stack tree automata are labelled by states of stack automata which have a further nested structure [5]. These automata are based on a similar automata model by Bouajjani and Meyer [2]. We give the formal definition with intuition following.

Definition 3.4 (Order-n Annotated Stack Tree Automata)**.** *An* order-n stack tree automaton *over a given stack alphabet* Σ *and set of control states* $\mathbb{P}$ *is a tuple*

$$\mathcal{T} = (\mathbb{Q}, \mathbb{R}_n, \ldots, \mathbb{R}_1, \Sigma, \Delta, \Delta_n, \ldots, \Delta_1, \mathbb{P}, \mathbb{F}, \mathbb{F}_n, \ldots, \mathbb{F}_1)$$

where Σ is a finite stack alphabet, $\mathbb{Q}$ is a finite set of states,

$$\Delta \subset \mathbb{Q} \times \{(i, m) \mid 1 \leq i \leq m\} \times (\mathbb{Q} \setminus \mathbb{F}) \times \mathbb{R}_n$$

is a finite set of transitions, $\mathbb{P} \subseteq \mathbb{Q}$ and $\mathbb{F} \subseteq \mathbb{Q}$ are initial and final states respectively, and

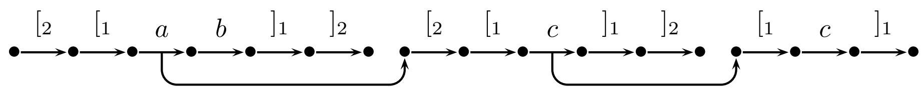
1. *for all $n \geq k \geq 2$, we have $\mathbb{R}_k$ is a finite set of states, $\Delta_k \subseteq \mathbb{R}_k \times \mathbb{R}_{k-1} \times 2^{\mathbb{R}_k}$ is a transition relation, and $\mathbb{F}_k \subseteq \mathbb{R}_k$ is a set of accepting states, and*

2. *$\mathbb{R}_1$ is a finite set of states, $\Delta_1 \subseteq \bigcup_{2 \leq k \leq n} \left(\mathbb{R}_1 \times \Sigma \times 2^{\mathbb{R}_k} \times 2^{\mathbb{R}_1}\right)$ is a transition relation, and $\mathbb{F}_1 \subseteq \mathbb{R}_1$ is a set of accepting states.*

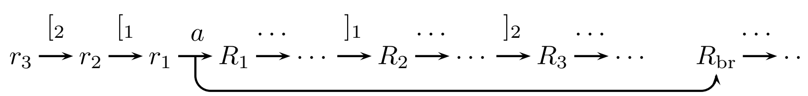
3.3.1 Accepting Stacks

Order-k stacks are recognised from states in $\mathbb{R}_k$. A transition $(r, r', R) \in \Delta_k$ from r to R for some $k > 1$ is denoted $r \xrightarrow{r'} R$ and can be fired when the stack is $s :_k s'$ and s is accepted from $r' \in \mathbb{R}_{(k-1)}$. The remainder of the stack s' must be accepted from all states in R. At order-1, a transition $(r, a, R_{\text{br}}, R) \in \Delta_1$ is denoted $r \xrightarrow[R_{\text{br}}]{a} R$ and is a standard alternating a-transition with the additional requirement that the stack annotating a is accepted from all states in R_{br}. A stack is accepted if a subset of $\mathbb{F}_k$ is reached at the end of each order-k stack. Note, we give a more formal definition of a run in Appendix A. We write $s \in \mathcal{L}_r(\mathcal{T})$ whenever s is accepted from a state r.

An order-n stack can be represented naturally as an edge-labelled tree over the alphabet $\{[_{n-1}, \ldots, [_1,]_1, \ldots,]_{n-1}\} \uplus \Sigma$, with Σ-labelled edges having a second target to the tree representing the annotation. For technical convenience, a tree representing an order-k stack does not use $[_k$ or $]_k$ symbols (these appear uniquely at the beginning and end of the stack). An example order-3 stack is given below, with only a few annotations shown. The annotations are order-3 and order-2 respectively.



An example (partial) run over this stack is pictured below, using transitions $r_3 \xrightarrow{r_2} R_3 \in \Delta_3$, $r_2 \xrightarrow{r_1} R_2 \in \Delta_2$, and $r_1 \xrightarrow[R_{\text{br}}]{a} R_1 \in \Delta_1$. The node labelled R_{br} begins a run on the stack annotating a.



3.3.2 Accepting Stack Trees

Annotated stack tree automata are bottom-up tree automata whose transitions are labelled by states from which stacks are accepted. We denote by

$$q \leftarrow_{i/m} (q', r)$$

a transition $(q, i, m, q', r) \in \Delta$. Observe that $q' \notin \mathbb{F}$ by definition. When a node v has children $v_1, \ldots, v_m$, the transition above could be applied to the ith child v_i. It can be applied when v_i

is already labelled by q' and the stack s_i attached to v_i is accepted from state r of the stack automaton. If it is applied, then q will be set as the label of the parent v. Over runs of the automaton we enforce that every child is present and the transitions applied at each child agree on the state assigned to its parent.

Let $\lambda_s(v) = s$ when $\lambda(v) = (p, s)$ or $\lambda(v) = s$. Given an order-n annotated stack tree $t = (\mathcal{D}, \lambda)$ a run of an automaton $\mathcal{T}$ is a $\mathbb{Q}$-labelled tree $(\mathcal{D}, \lambda')$ where each leaf v of t has $\lambda'(v) = p$ whenever $\lambda(v) = (p, s)$ for some s, and each internal node v with children $v1, \ldots, vm$ has a label $\lambda'(v) = q$ only if we have transitions

$$q \leftarrow_{1/m} (q_1, r_1), \ldots, q \leftarrow_{m/m} (q_m, r_m),$$

and $\lambda'(vi) = q_i$ and $\lambda_s(vi) \in \mathcal{L}_{r_i}(\mathcal{T})$ for all $1 \leq i \leq m$. Finally $\lambda'(\varepsilon) = q$ and we have a transition $q_f \leftarrow_{1/1} (q, r)$ with $q_f \in \mathbb{F}$ and $\lambda_s(\varepsilon) \in \mathcal{L}_r(\mathcal{T})$.

We write $\mathcal{L}(\mathcal{T})$ to denote the set of trees accepted by $\mathcal{T}$.

3.4 Notation and Conventions

3.4.1 Number of Transitions

We assume for all pairs of states $q, q' \in \mathbb{Q}$ and each i, m there is at most one transition of the form $q \leftarrow_{i/m} (q', r)$. Similarly we assume for all $r \in \mathbb{R}_k$ and $R \subseteq \mathbb{R}_k$ that there is at most one transition of the form $r \xrightarrow{r'} R \in \Delta_k$. This condition can easily be ensured by replacing pairs of transitions $r \xrightarrow{r_1} R$ and $r \xrightarrow{r_2} R$ with a single transition $r \xrightarrow{r'} R$, where r' accepts the union of the languages of stacks accepted from r_1 and r_2. Similarly for transitions in Δ.

3.4.2 Short-form Notation

Consider the example run shown above. This run reads the top of every level of the stack: the transition to R_3 reads the topmost order-2 stack, the transition to R_2 reads the order-1 stack at the top of this stack, and the transition to R_1 and R_{br} reads the top character of the order-1 stack.

The saturation algorithm relies on stack updates only affecting the topmost part of the stack. Thus, we need a notation for talking about the beginning of the run. Hence, we will write the run in the figure above (that reads the topmost parts of the stack) as a "short-form" transition

$$r_3 \xrightarrow[R_{\mathrm{br}}]{a} (R_1, \ldots, R_3) \ .$$

In the following, we define this notation formally, and generalise it to transitions of a stack tree automaton. In general, we write

$$r \xrightarrow[R_{\mathrm{br}}]{a} (R_1, \ldots, R_k) \text{ and } r \xrightarrow{r'} (R_{k'+1}, \ldots, R_k).$$

In the first case, $r \in \mathbb{R}_k$ and there exist $r_{k-1}, \ldots, r_1$ such that $r \xrightarrow{r_{k-1}} R_k \in \Delta_k$, $r_{k-1} \xrightarrow{r_{k-2}} R_{k-1} \in \Delta_{k-1}$, $\ldots$, $r_1 \xrightarrow[R_{\mathrm{br}}]{a} R_1 \in \Delta_1$. Since we assume at most one transition between any state and set of states, the intermediate states $r_{k-1}, \ldots, r_1$ are uniquely determined by r, a, R_{br} and $R_1, \ldots, R_k$.

In the second case, either $k = k'$ and $r = r' \in \mathbb{R}_k$, or $k > k'$ and we have $r \in \mathbb{R}_k$, $r' \in \mathbb{R}_{k'}$, and there exist $r_{k-1}, \ldots, r_{k'+1}$ with $r \xrightarrow{r_{k-1}} R_k \in \Delta_k$, $r_{k-1} \xrightarrow{r_{k-2}} R_{k-1} \in \Delta_{k-1}$, $\ldots$, $r_{k'+2} \xrightarrow{r_{k'+1}} R_{k'+2} \in \Delta_{k'+2}$ and $r_{k'+1} \xrightarrow{r'} R_{k'+1} \in \Delta_{k'+1}$.

We lift the short-form transition notation to transitions from sets of states. We assume that state-sets $\mathbb{R}_n, \ldots, \mathbb{R}_1$ are disjoint. Suppose $R = \{r_1, \ldots, r_m\}$ and for all $1 \leq i \leq m$ we have $r_i \xrightarrow[R^i_{\mathrm{br}}]{a} \left(R^i_1, \ldots, R^i_k\right)$. Then we have $R \xrightarrow[R_{\mathrm{br}}]{a} (R_1, \ldots, R_k)$ where $R_{\mathrm{br}} = \bigcup_{1 \leq i \leq m} R^i_{\mathrm{br}}$ and for all k, $R_k = \bigcup_{1 \leq i \leq m} R^i_k$. Because an annotation can only be of one order, we insist that $R_{\mathrm{br}} \subseteq \mathbb{R}_k$ for some k.

We generalise this to trees as follows. We write

$$q \leftarrow_{i/m} (q', a, R_{\mathrm{br}}, R_1, \ldots, R_n) \quad \text{and} \quad q \leftarrow_{i/m} (q', r', R_{k+1}, \ldots, R_n)$$

when $q \leftarrow_{i/m} (q', r)$ and $r \xrightarrow[R_{\mathrm{br}}]{a} (R_1, \ldots R_n)$ or, respectively, $r \xrightarrow{r'} (R_{k+1}, \ldots R_n)$.

Finally, we remark that a transition to the empty set is distinct from having no transition.

4 Backwards Reachability Analysis

Fix a GASTRS $\mathcal{G}$ and automaton $\mathcal{T}_0$ for the remainder of the article. We define

$$\mathrm{Pre}^*_{\mathcal{G}}(\mathcal{T}_0) = \{t \mid t \to^* t' \wedge t' \in \mathcal{L}(\mathcal{T}_0)\} \ .$$

We give a saturation algorithm for computing an automaton $\mathcal{T}$ such that $\mathcal{L}(\mathcal{T}) = \mathrm{Pre}^*_{\mathcal{G}}(\mathcal{T}_0)$. Indeed, we prove the following theorem. The upper bound is discussed in the sequel. The lower bound comes from alternating higher-order pushdown automata [8] and appears in Appendix D.

Theorem 4.1. *Given an order-n GASTRS $\mathcal{G}$ and stack tree automaton $\mathcal{T}_0$, $\mathrm{Pre}^*_{\mathcal{G}}(\mathcal{T}_0)$ is regular and computable in n-EXPTIME, which is optimal.*

For technical reasons assume for each p there is at most one rule $(p_1, \ldots, p_m) \xrightarrow{\bar{-}} p$. E.g., we cannot have $(p_1, p_2) \xrightarrow{\bar{-}} p$ and $(p'_1, p'_2) \xrightarrow{\bar{-}} p$. This is not a real restriction since we can introduce intermediate control states. E.g. $(p_1, p_2) \xrightarrow{\bar{-}} p_{1,2}$ and $p_{1,2} \xrightarrow{\mathrm{rew}_{a \to a}} p$ and $(p'_1, p'_2) \xrightarrow{\bar{-}} p'_{1,2}$ and $p'_{1,2} \xrightarrow{\mathrm{rew}_{a \to a}} p$ for all $a \in \Sigma$.

Initial States

We say that all states in $\mathbb{P}$ are *initial*. Furthermore, a state r is initial if there is a transition $q \leftarrow_{i/m} (q', r)$ or if there exists a transition $r' \xrightarrow{r} R$ in some Δ_k. We make the assumption that all initial states do not have any incoming transitions and that they are not final[1]. Furthermore, we assume any initial state only appears on one transition.

New Transitions

When we add a transition $q \leftarrow_{i/m} (q', a, R_{\mathrm{br}}, R_1, \ldots, R_n)$ to the automaton, then, we add $q \leftarrow_{i/m} (q', r_n)$ to Δ if it does not exist, else we use the existing r_n, and then for each $n \geq k > 1$, we add $r_k \xrightarrow{r_{k-1}} R_k$ to Δ_k if a transition between r_k and R_k does not already exist, otherwise we use the existing transition and state r_{k-1}; finally, we add $r_1 \xrightarrow[R_{\mathrm{br}}]{a} R_1$ to Δ_1.

[1] Hence automata cannot accept empty stacks from initial states. This can be overcome by introducing a bottom-of-stack symbol.

The Algorithm

We give the algorithm formally here, with intuitive explanations given in the follow section. Saturation is a fixed point algorithm. We begin with a GASTRS $\mathcal{G} = (\Sigma, \mathcal{R})$ and target set of trees by $\mathcal{T}_0$. Then, we apply the saturation function $\mathcal{F}$ and obtain a sequence of automata $\mathcal{T}_{i+1} = \mathcal{F}(\mathcal{T}_i)$. The algorithm terminates when $\mathcal{T}_{i+1} = \mathcal{T}_i$ in which case we will have $\mathcal{L}(\mathcal{T}_{i+1}) = \mathrm{Pre}^*_{\mathcal{G}}(\mathcal{T}_0)$.

Following the conventions described above for adding transitions to the automaton, we can only add a finite number of states to the automaton, which implies that only a finite number of transitions can be added. Hence, we must necessarily reach a fixed point for some i.

Given $\mathcal{T}_i$, we define $\mathcal{T}_{i+1} = \mathcal{F}(\mathcal{T}_i)$ to be the automaton obtained by adding to $\mathcal{T}_i$ the following transitions and states.

- For each rule $p \xrightarrow{\mathrm{rew}_{a \to b}} p' \in \mathcal{R}$ and transition $q \leftarrow_{j/m} (p', b, R_{\mathrm{br}}, R_1, \ldots, R_n)$ in $\mathcal{T}_i$, add to $\mathcal{T}_{i+1}$ the transition $q \leftarrow_{j/m} (p, a, R_{\mathrm{br}}, R_1, \ldots, R_n)$.

- For each rule $p \xrightarrow{\mathrm{push}^k_1} p' \in \mathcal{R}$, transition $q \leftarrow_{j/m} (p', a, R_{\mathrm{br}}, R_1, \ldots, R_n)$, and $R_1 \xrightarrow[R'_{\mathrm{br}}]{a} R'_1$ in $\mathcal{T}_i$, add
$$q \leftarrow_{j/m} (p, a, R'_{\mathrm{br}}, R'_1, R_2, \ldots, R_{k-1}, R_k \cup R_{\mathrm{br}}, R_{k+1}, \ldots, R_n)$$
to $\mathcal{T}_{i+1}$ when $k > 1$, and $q \leftarrow_{j/m} (p, a, R'_{\mathrm{br}}, R'_1 \cup R_{\mathrm{br}}, R_2, \ldots, R_n)$ when $k = 1$.

- For each rule $p \xrightarrow{\mathrm{push}_k} p' \in \mathcal{R}$ and $q \leftarrow_{j/m} (p', a, R_{\mathrm{br}}, R_1, \ldots, R_n)$ and $R_k \xrightarrow[R'_{\mathrm{br}}]{a} (R'_1, \ldots, R'_k)$ in $\mathcal{T}_i$, add to $\mathcal{T}_{i+1}$
$$q \leftarrow_{j/m} \left(p, a, R_{\mathrm{br}} \cup R'_{\mathrm{br}}, R_1 \cup R'_1, \ldots, R_{k-1} \cup R'_{k-1}, R'_k, R_{k+1}, \ldots, R_n\right) \ .$$

- For each rule $p \xrightarrow{\mathrm{pop}_k} p' \in \mathcal{R}$ and $q \leftarrow_{j/m} (p', r_k, R_{k+1}, \ldots, R_n)$ in $\mathcal{T}_i$, add to $\mathcal{T}_{i+1}$ for each $a \in \Sigma$
$$q \leftarrow_{j/m} (p, a, \emptyset, \emptyset, \ldots, \emptyset, \{r_k\}, R_{k+1}, \ldots, R_n) \ .$$

- For each rule $p \xrightarrow{\mathrm{collapse}_k} p' \in \mathcal{R}$ and $q \leftarrow_{j/m} (p', r_k, R_{k+1}, \ldots, R_n)$ in $\mathcal{T}_i$, add to $\mathcal{T}_{i+1}$ for each $a \in \Sigma$
$$q \leftarrow_{j/m} (p, a, \{r_k\}, \emptyset, \ldots, \emptyset, R_{k+1}, \ldots, R_n) \ .$$

- For each rule $p \xrightarrow{+} (p_1, \ldots, p_m) \in \mathcal{R}$ and $q \leftarrow_{j/m'} (q', a, R_{\mathrm{br}}, R_1, \ldots, R_n)$ and
$$q' \leftarrow_{1/m} \left(p_1, a, R^1_{\mathrm{br}}, R^1_1, \ldots, R^1_n\right), \ldots, q' \leftarrow_{m/m} \left(p_2, a, R^2_{\mathrm{br}}, R^2_1, \ldots, R^2_n\right)$$
in $\mathcal{T}_i$, add to $\mathcal{T}_{i+1}$
$$q \leftarrow_{j/m'} (p, a, R'_{\mathrm{br}}, R'_1, \ldots, R'_n)$$
where $R'_{\mathrm{br}} = R_{\mathrm{br}} \cup R^1_{\mathrm{br}} \cup \cdots \cup R^m_{\mathrm{br}}$ and for all k, we have $R'_k = R_1 \cup R^1_k \cup \cdots \cup R^m_k$.

- For each rule $(p_1, \ldots, p_m) \xrightarrow{-} p \in \mathcal{R}$ and $a_1, \ldots, a_m \in \Sigma$ add to $\mathcal{T}_{i+1}$ the transitions $p \leftarrow_{j/m} (p_j, a_j, \emptyset, \emptyset, \ldots, \emptyset)$ for each $1 \leq j \leq m$.

4.0.3 Intuition of the Algorithm

Since rules may only be applied to the leaves of the tree, the algorithm works by introducing new initial transitions that are derived from existing initial transitions. Consider a tree t with a leaf node v labelled by $(b^{s_{\mathrm{br}}} :_1 s)$. Suppose this tree were already accepted by the automaton, and the initial transition $q \leftarrow_{i/m} (p, b, R_{\mathrm{br}}, R_1, \ldots, R_n)$ is applied to v.

If we had a rule $p' \xrightarrow{\mathrm{rew}_{a \to b}} p$ then we could apply this rule to a tree t' that is identical to t except v is labelled by $(a^{s_{\mathrm{br}}} :_1 s)$. After the application, we would obtain t. Thus, if t is accepted by the automaton, then t' should be accepted.

The saturation algorithm will derive from the above rule and transition a new transition $q \leftarrow_{i/m} (p', b, R_{\mathrm{br}}, R_1, \ldots, R_n)$. This transition simply changes the control state and top character of the stack. Thus, we can substitute this transition into the accepting run of t to build an accepting run of t'.

For a rule $(p_1) \xrightarrow{-} p$ we would introduce a transition $p \leftarrow_{1/1} (b, p_1, \emptyset, \emptyset, \ldots, \emptyset)$. We can add this transition to any accepting run of a tree with a leaf with control state p and it will have the effect of adding a new node with control state p_1. Since we can obtain the original tree by applying the rule, the extended tree should also be accepted. The intuition is similar for the pop_k and $\mathrm{collapse}_k$ operations.

To understand the intuition for the push_k, push_1^k and $p \xrightarrow{+} (p_1, \ldots, p_m)$ rules, one must observe that these rules, applied backwards, have the effect of replacing multiple copies of identical stacks with a single stack. Thus, the new transitions accept the intersection of the stacks that could have been accepted by multiple previous transitions: taking the union of two sets of automaton states means that the intersection of the language must be accepted.

Correctness

We have the following property.

Property 4.1 (Correctness of Saturation). *Given an order-n GASTRS, saturation runs in n-EXPTIME and builds an automaton $\mathcal{T}$ such that $\mathcal{L}(\mathcal{T}) = \mathrm{Pre}^*_{\mathcal{G}}(\mathcal{T}_0)$.*

Proof. The proof of completeness is given in Lemma B.1 and soundness is given in Lemma C.6.

The complexity is derived as follows. We add at most one transition of the form $q \leftarrow_{i/m} (p, r)$ for each q, i, m and p. Hence we add at most a polynomial number of transitions to Δ.

Thus, to Δ_n we have a polynomial number of states. We add at most one transition of the form $r \xrightarrow{r'} R$ for each r and set of states R. Thus we have at most an exponential number of transitions in Δ_n.

Thus, in $\mathbb{R}_k$ we have a number of states bounded by a tower of exponentials of height $(n-k)$. Since we add at most one transition of the form $r \xrightarrow{r'} R$ for each r and R we have a number of transitions bounded by a tower of exponentials of height $(n - k + 1)$ giving the number of states in $\mathbb{R}_{k-1}$.

Thus, at order-1 the number of new transitions is bounded by a tower of height n, giving the n-EXPTIME complexity. □

5 Context Bounding

In the model discussed so far, communication between different nodes of the tree had to be done locally (i.e. from parent to child, via the destruction of nodes). We show that the saturation algorithm can be extended to allow a bounded amount of communication between distant nodes of the tree without destroying the nodes.

We begin by defining an extension of our model with global state. We then show that being able to compute $\mathrm{Pre}^*_{\mathcal{G}}(\mathcal{T}_0)$ can easily be adapted to allow a bounded number of global state changes.

5.1 GASTRS with Global State

Definition 5.1 (Order-n Ground Annotatee Stack Tree Rewrite Systems with Global State)**.** *An* order-n ground annotated stack tree rewrite system (GASTRS) with global state $\mathcal{G}$ *is a tuple* $(\Sigma, \mathbb{P}, \mathbb{G}, \mathcal{R})$ *where* Σ *is a finite stack alphabet,* $\mathbb{P}$ *is a finite set of control states,* $\mathbb{G}$ *is a finite set of global states, and* $\mathcal{R} \subset \mathbb{G} \times \mathrm{STOps}^{\Sigma,\mathbb{P}}_n \times \mathbb{G}$ *is a finite set of operations.*

A configuration of an order-n GASTRS with global state is a pair (g, t) where $g \in \mathbb{G}$ and t is an order-n annotated stack tree over alphabet Σ. We have a transition $(g, t) \to (g', t')$ whenever there is some $(g, \theta, g') \in \mathcal{R}$ and $t' \in \theta(t)$. We write $t \to^* t'$ when there is a run $t = t_0 \to \cdots \to t_m = t'$.

5.2 The Context-Bounded Reachability Problem

The context-bounded reachability problem is to compute the set of configurations from which there is a run to some target set of configurations, and moreover, the global state is only changed at most ι times, where ι is some bound given as part of the input.

Definition 5.2 (Global Context-Bounded Backwards Reachability Problem)**.** *Given a GASTRS with global state $\mathcal{G}$, and a stack tree automaton $\mathcal{T}^0_g$ for each $g \in \mathbb{G}$, and a bound ι, the* global context-bounded backwards reachability problem *is to compute a stack tree automaton $\mathcal{T}_g$ for each $g \in \mathbb{G}$, such that $t \in \mathcal{L}(\mathcal{T}_g)$ iff there is a run*

$$(g, t) = (g_0, t_0) \to \cdots \to (g_m, t_m) = (g', t')$$

with $t' \in \mathcal{L}\big(\mathcal{T}^0_{g'}\big)$ and there are at most ι transitions during the run such that $g_i \neq g_{i+1}$.

5.3 Decidability of Context-Bounded Reachability

Since the number of global state changes is bounded, the sequence of global state changes for any run witnessing context-bounded reachability is of the form $g_0, \ldots, g_m$ where $m \leq \iota$. Let $\tilde{\mathbb{G}}$ be the set of such sequences.

Suppose we could compute for each such sequence $\tilde{g} = g_0, \ldots, g_m$ an automaton $\mathcal{T}_{\tilde{g}}$ such that $t \in \mathcal{L}(\mathcal{T}_{\tilde{g}})$ iff there is a run from (g_0, t) to (g_m, t') with $t' \in \mathcal{L}(\mathcal{T}_{g_m})$ where the sequence of global states appearing on the run is $\tilde{g}$. We could then compute an answer to the global context-bounded backwards reachability problem by taking

$$\mathcal{T}_g = \bigcup_{g\tilde{g} \in \tilde{\mathbb{G}}} \mathcal{T}_{g\tilde{g}} \ .$$

To compute $\mathcal{T}_{\tilde{g}}$ we first make the simplifying assumption (without loss of generality) that for each $g \neq g'$ there is a unique $(g, \theta, g') \in \mathcal{R}$ and moreover $\theta = p \xrightarrow{\mathrm{rew}_{a \to b}} p'$. Furthermore, for all $g \in \mathbb{G}$ we define $\mathcal{G}_g = (\Sigma, \mathbb{P}, \mathcal{R}_g)$ where

$$\mathcal{R}_g = \{\theta \mid (g, \theta, g) \in \mathcal{R}\} \ .$$

We compute $\mathcal{T}_{\tilde{g}}$ by backwards induction. Initially, when $\tilde{g} = g$ we compute

$$\mathcal{T}_{\tilde{g}} = \mathrm{Pre}^*_{\mathcal{G}_g}(\mathcal{T}_g)\ .$$

It is immediate to see that $\mathcal{T}_{\tilde{g}}$ is correct. Now, assume we have $\tilde{g} = g\tilde{g}'$ and we have already computed $\mathcal{T}_{\tilde{g}'}$, we show how to compute $\mathcal{T}_{\tilde{g}}$.

The first step is to compute $\mathcal{T}'_{\tilde{g}}$ such that $t \in \mathcal{L}\big(\mathcal{T}'_{\tilde{g}}\big)$ iff $(g, t) \to (g', t')$ where g' is the first state of $\tilde{g}'$ and $t' \in \mathcal{L}(\mathcal{T}_{\tilde{g}'})$. That is, $\mathcal{T}'_{\tilde{g}}$ accepts all trees from which we can change the current global state to g'. That is, by a single application of the unique rule (g, θ, g'). Once we have computed this automaton we need simply build

$$\mathcal{T}_{\tilde{g}} = \mathrm{Pre}^*_{\mathcal{G}_g}\big(\mathcal{T}'_{\tilde{g}}\big)$$

and we are done.

We first define $\mathcal{T}''_{\tilde{g}}$ which is a version of $\mathcal{T}_{\tilde{g}'}$ that has been prepared for a single application of (g, θ, g'). From this we compute $\mathcal{T}_{\tilde{g}}$.

The strategy for building $\mathcal{T}''_{\tilde{g}}$ is to mark in the states which child, if any, of the node has the global state change rule applied to its subtree. At each level of the tree, this marking information enforces that only one subtree contains the application. Thus, when the root is reached, we know there is only one application in the whole tree. Note, this automaton does not contain any transitions corresponding to the actual application of the global change rule. This is added afterwards to compute $\mathcal{T}_{\tilde{g}}$. Thus, if

$$\mathcal{T}_{\tilde{g}} = (\mathbb{Q}, \mathbb{R}_n, \ldots, \mathbb{R}_1, \Sigma, \Delta, \Delta_n, \ldots, \Delta_1, \mathbb{P}, \mathbb{F}', \mathbb{F}_n, \ldots, \mathbb{F}_1)$$

then

$$\mathcal{T}''_{\tilde{g}} = (\mathbb{Q}', \mathbb{R}_n, \ldots, \mathbb{R}_1, \Sigma, \Delta', \Delta_n, \ldots, \Delta_1, \mathbb{P}, \mathbb{F}', \mathbb{F}_n, \ldots, \mathbb{F}_1)$$

where, letting m be the maximum number of children permitted by any transition of $\mathcal{T}_{\tilde{g}}$,

$$\mathbb{Q}' = \mathbb{P} \cup \mathbb{Q} \times \{0, \ldots, m\} \quad \text{and} \quad \mathbb{F}' = \{(q_f, i) \mid q_f \in \mathbb{F} \wedge 0 < i \leq m\}$$

and we define

$$\begin{array}{rcl}
\Delta' & = & \Delta_{\mathrm{init}} \cup \Delta_{\mathrm{noapp}} \cup \Delta_{\mathrm{pass}} \\[2ex]
\Delta_{\mathrm{init}} & = & \begin{array}{l} \big\{(q, 0) \leftarrow_{i/m} (p, r) \;\big|\; q \leftarrow_{i/m} (p, r) \in \Delta\big\} \cup \\ \big\{(q, j) \leftarrow_{i/m} (p, r) \;\big|\; q \leftarrow_{i/m} (p, r) \in \Delta \wedge i \neq j\big\} \end{array} \\[3ex]
\Delta_{\mathrm{noapp}} & = & \big\{(q, 0) \leftarrow_{i/m} ((q', 0), r) \;\big|\; q \leftarrow_{i/m} (q', r) \in \Delta\big\} \\[2ex]
\Delta_{\mathrm{pass}} & = & \begin{array}{l} \big\{(q, i) \leftarrow_{i/m} ((q, j), r) \;\big|\; q \leftarrow_{i/m} (q', r) \in \Delta\big\} \cup \\ \big\{(q, j) \leftarrow_{i/m} ((q, 0), r) \;\big|\; q \leftarrow_{i/m} (q', r) \in \Delta \wedge i \neq j\big\}\ . \end{array}
\end{array}$$

In the above Δ_{init} has two kinds of transitions. The first set are the initial transitions for the nodes to which the rewrite rule is not applied (indicated by the 0). The second set are the rules where the rewrite rule is applied at the jth sibling of the ith child. Next Δ_{noapp} are the transitions for subtrees which have not been marked as containing the application. Finally, Δ_{pass} propagates information about where the application actually occurred up the tree. The first set of transitions in Δ_{pass} are used when the ith child contains the application (hence it labels the parent with the information that the ith child contains the application). The second set of transitions guess that the jth sibling contains the application. Thus, at any node, at most one child subtree may contain the application. The set of final states enforce that the application has occurred in some child.

To compute $\mathcal{T}'_{\tilde{g}}$, letting $\theta = p \xrightarrow{\text{rew}_{a \to b}} p'$ be the operation on the global state change, we add to $\mathcal{T}''_{\tilde{g}}$ a transition

$$(q, i) \leftarrow_{i/m} (p, a, R_{\text{br}}, R_1, \ldots, R_n)$$

for each

$$q \leftarrow_{i/m} (p', b, R_{\text{br}}, R_1, \ldots, R_n)$$

in $\mathcal{T}_{\tilde{g}'}$.

We remark that, as defined, $\mathcal{T}_{\tilde{g}}$ does not satisfy the prerequisites of the saturation algorithm, since initial states reading stacks might have incoming transitions, and, moreover, an initial state may label more than one transition. We can convert $\mathcal{T}_{\tilde{g}}$ to the correct format using the automata manipulations in Appendix A.

Lemma 5.1. *We have $t \in \mathcal{L}\big(\mathcal{T}'_{\tilde{g}}\big)$ iff $(g, t) \to (g', t')$ via a single application of the transition (g, θ, g') and $t' \in \mathcal{L}(\mathcal{T}_{\tilde{g}'})$.*

Proof. First, assume $t \in \mathcal{L}\big(\mathcal{T}'_{\tilde{g}}\big)$. We argue that there is exactly one leaf $t_{\bullet_i}$ read by a transition $(q, i) \leftarrow_{i/m} (p, r)$ and all other leaves are read by some $(q, 0) \leftarrow_{i/m} (p, r)$ or $(q, j) \leftarrow_{i/m} (p, r)$ with $j \neq i$.

If there is no such $t_{\bullet_i}$ then all leaf nodes are read by some $(q, 0) \leftarrow_{i/m} (p, r)$. Thus, all parents of the leaf nodes are labelled by $(q, 0)$. Thus, take any node v and assume its children are labelled by some $(q, 0)$. It must be the case that v is also labelled by some $(q, 0)$ since otherwise it is labelled (q, i) and its ith child must be labelled by some (q, j) with $j > 0$, which is a contradiction. Hence, the accepting state of the run must also be some $(q_f, 0)$ which is not possible.

If there are two or more leaves labelled by some (q, i) with $i > 0$ then each ancestor must also be labelled by some (q, i) with $i > 0$. Take the nearest common ancestor v and suppose it is labelled (q, i). However, since it has two children labelled with non-zero second components, we must have used a transition $(q, i) \leftarrow_{j/m} ((q', j'), r)$ which, by definition, cannot exist.

Hence, we have only one leaf $t_{\bullet_i}$ where

$$(q, i) \leftarrow_{i/m} (p, a, R_{\text{br}}, R_1, \ldots, R_n)$$

is used. Obtain t' by applying $p \xrightarrow{\text{rew}_{a \to b}} p'$ at this leaf. We build an accepting run of $\mathcal{T}_{\tilde{g}'}$ by taking the run of $\mathcal{T}'_{\tilde{g}}$ over t, projecting out the second component of each label, and replacing the transition used at $t_{\bullet_i}$ with

$$q \leftarrow_{i/m} (p', b, R_{\text{br}}, R_1, \ldots, R_n) \ .$$

Hence, we are done.

In the other direction take t and t' obtained by applying $p \xrightarrow{\text{rew}_{a \to b}} p'$ at leaf $t_{\bullet_i}$. We take the accepting run of $\mathcal{T}_{\tilde{g}'}$ over t' and build an accepting run of $\mathcal{T}'_{\tilde{g}}$ over t. Let

$$q \leftarrow_{i/m} (p', b, R_{\text{br}}, R_1, \ldots, R_n) \ .$$

be the transition used at $t_{\bullet_i}$. We replace it with

$$(q, i) \leftarrow_{i/m} (p, a, R_{\text{br}}, R_1, \ldots, R_n) \ .$$

Starting from above the root node, let the jth child be the first on the path to $t_{\bullet_i}$ (the root node is the 1st child of "above the root node"). For all children except the jth, take the transition $q \leftarrow_{j'/m} (q', r)$ used in the run over t' and replace it with $(q, j) \leftarrow_{j'/m} ((q', 0), r)$. The remainder of the run in the descendents of these children requires us to use $(q, 0) \leftarrow_{i'/m} ((q', 0), r)$ or $(q, 0) \leftarrow_{i'/m} (q, r)$ instead of $q \leftarrow_{i'/m} (q', r)$.

For the jth child, we use instead of $q \leftarrow_{j/m} (q', r)$. the transition $(q, j) \leftarrow_{j/m} ((q', j'), r)$ when the j'th child of this child leads to $t_{\bullet_i}$ or the previously identified transition when the j'th child of this child is the leaf.

We repeat the routine above until we reach $t_{\bullet_i}$, at which point we've constructed an accepting run of $\mathcal{T}'_{\hat{g}}$ over t. □

By iterating the above procedure, we obtain our result.

Theorem 5.1 (Context-Bounded Reachability)**.** *The global context-bounded backwards reachability problem for GASTRS with global state is decidable.*

6 Conclusions and Future Work

We gave a saturation algorithm for annotated stack trees – a generalisation of annotated pushdown systems with the ability to fork and join threads. We build on the saturation method implemented by the C-SHORe tool. We would like to implement this work. We may also investigate higher-order versions of senescent ground tree rewrite systems [13], which generalises scope-bounding [22] to trees.

References

[1] A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model-checking. In *CONCUR*, pages 135–150, 1997.

[2] A. Bouajjani and A. Meyer. Symbolic Reachability Analysis of Higher-Order Context-Free Processes. In *Proc. 24rd Conf. on Found. of Software Technology and Theoretical Computer Science (FSTTCS'04)*, volume 3328 of *Lecture Notes in Computer Science*, Madras, India, December 2004. Springer Pub.

[3] A. Bouajjani, M. Müller-Olm, and T. Touili. Regular symbolic analysis of dynamic networks of pushdown systems. In *CONCUR 2005 - Concurrency Theory, 16th International Conference, CONCUR 2005, San Francisco, CA, USA, August 23-26, 2005, Proceedings*, pages 473–487, 2005.

[4] W. S. Brainerd. Tree generating regular systems. *Information and Control*, 14(2):217–231, 1969.

[5] C. H. Broadbent, A. Carayol, M. Hague, and O. Serre. A saturation method for collapsible pushdown systems. In *Automata, Languages, and Programming - 39th International Colloquium, ICALP 2012, Warwick, UK, July 9-13, 2012, Proceedings, Part II*, pages 165–176, 2012.

[6] C. H. Broadbent, A. Carayol, M. Hague, and O. Serre. C-shore: a collapsible approach to higher-order verification. In *ICFP*, pages 13–24, 2013.

[7] C. H. Broadbent and N. Kobayashi. Saturation-based model checking of higher-order recursion schemes. In *Computer Science Logic 2013 (CSL 2013), CSL 2013, September 2-5, 2013, Torino, Italy*, pages 129–148, 2013.

[8] T. Cachat and I. Walukiewicz. The complexity of games on higher order pushdown automata. *CoRR*, abs/0705.0262, 2007.

[9] A. Carayol and S. Wöhrle. The caucal hierarchy of infinite graphs in terms of logic and higher-order pushdown automata. In *FSTTCS*, pages 112–123, 2003.

[10] R. Chadha and M. Viswanathan. Decidability results for well-structured transition systems with auxiliary storage. In *CONCUR 2007 - Concurrency Theory, 18th International Conference, CONCUR 2007, Lisbon, Portugal, September 3-8, 2007, Proceedings*, pages 136–150, 2007.

[11] T. M. Gawlitza, P. Lammich, M. Müller-Olm, H. Seidl, and A. Wenner. Join-lock-sensitive forward reachability analysis for concurrent programs with dynamic process creation. In *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings*, pages 199–213, 2011.

[12] M. Hague. Saturation of concurrent collapsible pushdown systems. In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2013, December 12-14, 2013, Guwahati, India*, pages 313–325, 2013.

[13] M. Hague. Senescent ground tree rewrite systems. In *Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS '14, Vienna, Austria, July 14 - 18, 2014*, page 48, 2014.

[14] M. Hague, A. S. Murawski, C.-H. Luke Ong, and O. Serre. Collapsible pushdown automata and recursion schemes. In *LICS*, pages 452–461, 2008.

[15] T. Knapik, D. Niwinski, and P. Urzyczyn. Higher-order pushdown trees are easy. In *FoSSaCS '02: Proceedings of the 5th International Conference on Foundations of Software Science and Computation Structures*, pages 205–222, London, UK, 2002. Springer-Verlag.

[16] N. Kobayashi. Model-checking higher-order functions. In *PPDP*, pages 25–36, 2009.

[17] N. Kobayashi. Higher-order model checking: From theory to practice. In *LICS*, pages 219–224, 2011.

[18] N. Kobayashi. A practical l inear time algorithm for trivial automata model checking of higher-order recursion schemes. In *FOSSACS*, pages 260–274, 2011.

[19] N. Kobayashi and A. Igarashi. Model-checking higher-order programs with recursive types. In *ESOP*, pages 431–450, 2013.

[20] Naoki Kobayashi. GTRecS2: A model checker for recursion schemes based on games and types. A tool available at `http://www-kb.is.s.u-tokyo.ac.jp/~koba/gtrecs2/`, 2012.

[21] M. Kretínský, V. Rehák, and J. Strejcek. Extended process rewrite systems: Expressiveness and reachability. In *CONCUR 2004 - Concurrency Theory, 15th International Conference, London, UK, August 31 - September 3, 2004, Proceedings*, pages 355–370, 2004.

[22] S. La Torre and M. Napoli. Reachability of multistack pushdown systems with scope-bounded matching relations. In *CONCUR 2011 - Concurrency Theory - 22nd International Conference, CONCUR 2011, Aachen, Germany, September 6-9, 2011. Proceedings*, pages 203–218, 2011.

[23] A. W. Lin. Weakly-synchronized ground tree rewriting - (with applications to verifying multi-threaded programs). In *Mathematical Foundations of Computer Science 2012 - 37th International Symposium, MFCS 2012, Bratislava, Slovakia, August 27-31, 2012. Proceedings*, pages 630–642, 2012.

[24] C. Löding. *Infinite Graphs Generated by Tree Rewriting*. PhD thesis, RWTH Aachen, 2003.

[25] D. Lugiez and P. Schnoebelen. The regular viewpoint on pa-processes. In *CONCUR '98: Concurrency Theory, 9th International Conference, Nice, France, September 8-11, 1998, Proceedings*, pages 50–66, 1998.

[26] A. N. Maslov. Multilevel stack automata. *Problems of Information Transmission*, 15:1170–1174, 1976.

[27] R. P. Neatherway, S. J. Ramsay, and C.-H. L. Ong. A traversal-based algorithm for higher-order model checking. In *ICFP*, pages 353–364, 2012.

[28] C.-H. L. Ong. On model-checking trees generated by higher-order recursion schemes. In *LICS*, pages 81–90, 2006.

[29] P. Parys. Collapse operation increases expressive power of deterministic higher order pushdown

automata. In *STACS*, pages 603–614, 2011.

[30] V. Penelle. Rewriting higher-order stack trees, 2015. arXiv:1311.4915 [cs.FL], to appear in CSR 15.

[31] S. J. Ramsay, R. P. Neatherway, and C.-H. Luke Ong. A type-directed abstraction refinement approach to higher-order model checking. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 61–72, 2014.

[32] S. Qadeer and J. Rehof. Context-bounded model checking of concurrent software. In *TACAS*, pages 93–107, 2005.

[33] S. Salvati and I. Walukiewicz. Krivine machines and higher-order schemes. In *ICALP (2)*, pages 162–173, 2011.

[34] S. Schwoon. *Model-checking Pushdown Systems*. PhD thesis, Technical University of Munich, 2002.

[35] A. Seth. Games on higher order multi-stack pushdown systems. In *RP*, pages 203–216, 2009.

[36] D. Suwimonteerabuth, F. Berger, S. Schwoon, and J. Esparza. jmoped: A test environment for java programs. In *CAV*, pages 164–167, 2007.

[37] K. Yasukata, N. Kobayashi, and K. Matsuda. Pairwise reachability analysis for higher order concurrent programs by higher-order model checking. In *CONCUR 2014 - Concurrency Theory - 25th International Conference, CONCUR 2014, Rome, Italy, September 2-5, 2014. Proceedings*, pages 312–326, 2014.

A Particulars of Annotated Stack Tree Automata

Here we discuss various particulars of our stack tree automata: the definition of runs, the effective boolean algebra, membership, emptiness, transformations to normal form, and comparisons with other possible stack tree automata definitions.

A.1 Definition of Runs over Stacks

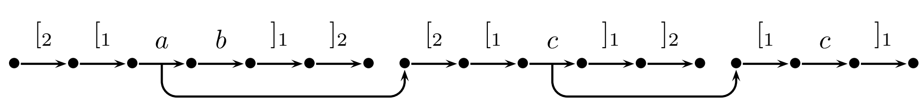
We give a more formal definition of a run accepting a stack. First we introduce some notation.

For $n \geq k > 1$, we write $R_1 \xrightarrow{R'} R_2$ to denote an order-k transition from a set of states whenever $R_1 = \{r_1, \ldots, r_m\}$ and for each $1 \leq i \leq m$ we have $r_i \xrightarrow{r'_i} R_i$ and $R' = \{r'_1, \ldots, r'_m\}$ and $R_2 = \bigcup_{1 \leq i \leq m} R_i$. The analogous notation at order-1 is a special case of the short-form notation defined in Section 3.4.

Formally, fix an annotated stack tree automaton

$$\mathcal{T} = (\mathbb{Q}, \mathbb{R}_n, \ldots, \mathbb{R}_1, \Sigma, \Delta, \Delta_n, \ldots, \Delta_1, \mathbb{P}, \mathbb{F}, \mathbb{F}_n, \ldots, \mathbb{F}_1)$$

We say a node *contains* a character if its exiting edge is labelled by the character. Recall the tree view of an annotated stack, an example of which is given below.



Some stack (tree) s is accepted by $\mathcal{T}$ from states $R_0 \subseteq \mathbb{R}_k$ — written $s \in \mathcal{L}_{R_0}(\mathcal{T})$ — whenever the nodes of the tree can be labelled by elements of $\bigcup\limits_{1 \leq k' \leq n} 2^{\mathbb{R}_{k'}}$ such that

1. R_0 is a subset of the label of the node containing the first $[_{k-1}$ character of the word, or if $k = 1$, the first character $a \in \Sigma$, and

2. for any node containing a character $[_{k'}$ labelled by R, then for all $r_1 \in R$, there exists some transition $(r_1, r_2, R_1) \in \Delta_{k'+1}$ such that r_2 appears in the label of the succeeding node and R_1 is a subset of the label of the node succeeding the matching $]_{k'}$ character, and

3. for any node containing a character $]_{k'}$, the label R is a subset of $\mathbb{F}_{k'}$, and the final node of an order-k stack is labelled by $R \subseteq \mathbb{F}_k$, and

4. for any node containing a character $a \in \Sigma$, labelled by R, for all $r' \in R$, there exists some transition $(r', a, R_{\mathrm{br}}, R') \in \Delta_1$ such that R_{br} is a subset of the label of the node annotating a, and R' is a subset of the label of the succeeding node.

That is, a stack automaton is essentially a stack- and annotation-aware alternating automaton, where annotations are treated as special cases of the alternation.

A.2 Effective Boolean Algebra

In this section we prove the following.

Proposition A.1. *Annotated stack tree automata form an effective boolean algebra.*

Proof. This follows from Proposition A.2, Proposition A.3, and Proposition A.4 below. ☐

Proposition A.2. *Given two automata*

$$\mathcal{T} = (\mathbb{Q}, \mathbb{R}_n, \ldots, \mathbb{R}_1, \Sigma, \Delta, \Delta_n, \ldots, \Delta_1, \mathbb{P}, \mathbb{F}, \mathbb{F}_n, \ldots, \mathbb{F}_1)$$

and

$$\mathcal{T}' = (\mathbb{Q}', \mathbb{R}'_n, \ldots, \mathbb{R}'_1, \Sigma, \Delta', \Delta'_n, \ldots, \Delta'_1, \mathbb{P}', \mathbb{F}', \mathbb{F}'_n, \ldots, \mathbb{F}'_1)$$

there is an automaton $\mathcal{T}''$ which recognises the union of the languages of $\mathcal{T}$ and $\mathcal{T}'$.

Proof. Supposing $\mathcal{T}$ and $\mathcal{T}'$ are disjoint except for $\mathbb{P}$ and no state $p \in \mathbb{P}$ has any incoming transition, the automaton we construct is:

$$\mathcal{T}'' = \left(\begin{array}{l} \mathbb{Q} \cup \mathbb{Q}', \\ \mathbb{R}_n \cup \mathbb{R}'_n, \ldots, \mathbb{R}_1 \cup \mathbb{R}'_1, \\ \Sigma, \\ \Delta \cup \Delta', \Delta_n \cup \Delta'_n, \ldots, \Delta_1 \cup \Delta'_1, \\ \mathbb{P}, \\ \mathbb{F} \cup \mathbb{F}', \mathbb{F}_n \cup \mathbb{F}'_n, \ldots, \mathbb{F}_1 \cup \mathbb{F}'_1 \end{array} \right)$$

Every run in $\mathcal{T}$ (resp $\mathcal{T}'$) is a run of $\mathcal{T}''$ as every state and transition of $\mathcal{T}$ is in $\mathcal{T}''$.

A run in $\mathcal{T}''$ is a run of $\mathcal{T}$ or of $\mathcal{T}'$, as every state and transition $\mathcal{T}''$ is in $\mathcal{T}$ or in $\mathcal{T}'$, and as the sets of states and transitions are disjoint except for initial states (which do not have incoming transitions), a valid run is either entirely in $\mathcal{T}$ or in $\mathcal{T}'$. ☐

Proposition A.3. *Given two automata*

$$\mathcal{T} = (\mathbb{Q}, \mathbb{R}_n, \ldots, \mathbb{R}_1, \Sigma, \Delta, \Delta_n, \ldots, \Delta_1, \mathbb{P}, \mathbb{F}, \mathbb{F}_n, \ldots, \mathbb{F}_1)$$

and

$$\mathcal{T}' = (\mathbb{Q}', \mathbb{R}'_n, \ldots, \mathbb{R}'_1, \Sigma, \Delta', \Delta'_n, \ldots, \Delta'_1, \mathbb{P}', \mathbb{F}', \mathbb{F}'_n, \ldots, \mathbb{F}'_1)$$

there is an automaton $\mathcal{T}''$ which recognises the intersection of the languages of $\mathcal{T}$ and $\mathcal{T}'$.

Proof. We construct the following automaton:

$$\mathcal{T}' = (\mathbb{Q}'', \mathbb{R}''_n, \ldots, \mathbb{R}''_1, \Sigma, \Delta'', \Delta''_n, \ldots, \Delta''_1, \mathbb{P}'', \mathbb{F}'', \mathbb{F}''_n, \ldots, \mathbb{F}''_1)$$

For any pair of states $r, r' \in \mathbb{R}_n \cup \mathbb{R}'_n$ we can assume a state $r \cap r'$ accepting the intersection of the stacks accepted from r and r'. This comes from the fact that stack automata form an effective boolean algebra [5]. The states and transitions in $\mathbb{R}''_n, \ldots, \mathbb{R}''_1$, $\Delta''_n, \ldots, \Delta''_1$, and $\mathbb{F}''_n, \ldots, \mathbb{F}''_1$ come from this construction.

For $q_1 \in \mathbb{Q}$ and $q_2 \in \mathbb{Q}'$, we define $q_{1,2}$ to be in $\mathbb{Q}''$ such that, for every $q_1 \leftarrow_{i/m} (q'_1, r_1)$ and $q_2 \leftarrow_{i/m} (q'_2, r_2)$, we add the transition $q_{1,2} \leftarrow_{i/m} \left(q'_{1,2}, r_1 \cap r_2\right)$.

We have $q_{1,2} \in \mathbb{F}''$ if and only if $q_1 \in \mathbb{F}$ and $q_2 \in \mathbb{F}'$.

A run exists in $\mathcal{T}''$ if and only if there is a run in $\mathcal{T}$ and one in $\mathcal{T}'$, by construction. □

Proposition A.4. *Given an automaton,*

$$\mathcal{T} = (\mathbb{Q}, \mathbb{R}_n, \ldots, \mathbb{R}_1, \Sigma, \Delta, \Delta_n, \ldots, \Delta_1, \mathbb{P}, \mathbb{F}, \mathbb{F}_n, \ldots, \mathbb{F}_1)$$

there is an automaton $\mathcal{T}'$ which accepts a tree if and only if it is not accepted by $\mathcal{T}$.

Proof. We define the complement as follows. We first assume that for each $r \in \mathbb{R}_n$ we also have $\overline{r} \in \mathbb{R}_n$ that accepts the complement of r. This follows from the complementation of stack automata in ICALP 2012 [5].

Then, we define $\mathcal{T}'$ to be the complement of $\mathcal{T}$, which contains

$$\mathcal{T}' = (\mathbb{Q}', \mathbb{R}_n, \ldots, \mathbb{R}_1, \Sigma, \Delta', \Delta_n, \ldots, \Delta_1, \mathbb{P}, \mathbb{F}', \mathbb{F}_n, \ldots, \mathbb{F}_1)$$

where, letting $m_{\max}$ be the maximum number of children that can appear in a tree accepted by $\mathcal{T}$ (this information is easily obtained from the transitions of $\mathcal{T}$), we have

$$\mathbb{Q}' = \bigcup_{m \leq m_{\max}} \left(2^{\mathbb{Q}}\right)^m .$$

That is, the automaton will label nodes of the tree with a set of states for each child. The ith set will be the set of all labels q that could have come from the ith child in a run of $\mathcal{T}$. Since all children have to agree on the q that labels a node, then a label $(Q_1, \ldots, Q_m)$ means that the set $Q_1 \cap \cdots \cap Q_m$ is the set of states q that could have labelled the node in a run of $\mathcal{T}$.

The transition relation Δ' is the set of transitions of the form

$$(Q_1, \ldots, Q_m) \leftarrow_{i/m} ((Q'_1, \ldots, Q'_{m'}), r)$$

where $m, m' \leq m_{\max}$ and for all $j \neq i$, the set Q_j is any subset of $\mathbb{Q}$, and $Q_i \subseteq \mathbb{Q}$ and r are such that

- $r = \bigcap\limits_{q \in \mathbb{Q}} r_q$, and

- if $q \in Q_i$ then
$$r_q = r_1 \cup \cdots \cup r_l$$
where $q \leftarrow_{i/m} (q_1, r_1)$, ..., $q \leftarrow_{i/m} (q_l, r_l)$ are all transitions to q via the ith of m children with the property that
$$q_j \in Q'_1 \cap \cdots \cap Q'_{m'}$$
for all j.

- if $q \notin Q_i$ then
$$r_q = \overline{r_1} \cap \cdots \cap \overline{r_l}$$
where $q \leftarrow_{i/m} (q_1, r_1)$, ..., $q \leftarrow_{i/m} (q_l, r_l)$ are all transitions to q via the ith of m children with the property that
$$q_j \in Q'_1 \cap \cdots \cap Q'_{m'}$$
for all j.

In each transition, the sets Q_j for all $j \neq i$ have no constraints. The automaton effectively guesses the set of labels that could have come from sibling nodes. The set Q_i contains all labellings that could have come from the ith child given the set of labellings that could have labelled the child. The final condition above insists that transitions to any state not in Q_i could not have been applied to the child.

The set of accepting states is
$$\{(Q_1, \ldots, Q_m) \mid \nexists q_f \in \mathcal{F}.q_f \in Q_1 \cap \cdots \cap Q_m\} \ .$$
For the initial states, we alias $p = \{p\}$

We prove that this automaton is the complement of $\mathcal{T}$. Associate to each node v the set Q_v such that $q \in Q_v$ iff there is some (partial, starting from the leaves) run of $\mathcal{T}$ that labels v with q. We prove that all runs of $\mathcal{T}'$ label v with some $(Q_1, \ldots, Q_m)$ such that $Q_v = Q_1 \cap \cdots \cap Q_m$.

At the leaves of the tree this is immediate since $\mathcal{T}$ must label the node with some p, and $\mathcal{T}'$ must label it with $\{p\}$.

Now, suppose we have a node v with children $v1$, ..., vm and the property holds for all children.

Take some $q \in Q_v$. Let $q \leftarrow_{1/m} (q_1, r_1)$, ..., $q \leftarrow_{m/m} (q_m, r_m)$ be the transitions used in the run labelling v with q. For each i we must have by induction q_i appearing in all sets labelling vi in a run of $\mathcal{T}'$. Now suppose $\mathcal{T}'$ labels v with $(Q_1, \ldots, Q_m)$ and moreover $q \notin Q_i$. Then, by construction, we must have that the stack labelling vi is accepted from $\overline{r_i}$. However, since the stack must have been accepted from r_i we have a contradiction. Thus, $q \in Q_i$.

Now take some $q \notin Q_v$. Thus, there is some i such that, letting $q \leftarrow_{i/m} (q_1, r_1)$, ..., $q \leftarrow_{i/m} (q_l, r_l)$ be all transitions with q_j appearing in Q_{vi}, we know the stack labelling vi is not accepted from any r_j (and is accepted from all $\overline{r_j}$). Now suppose $\mathcal{T}'$ labels v with $(Q_1, \ldots, Q_m)$ and moreover $q \in Q_i$. Then, by construction, we must have that the stack labelling vi is accepted from some r_j, which is a contradiction. Thus, $q \notin Q_i$.

Hence $Q_v = Q_1 \cap \cdots \cap Q_m$ as required.

Now, assume there is some accepting run of $\mathcal{T}$ via final state q_f. Assume there is an accepting run of $\mathcal{T}'$. Then necessarily the run of $\mathcal{T}'$ has as its final label some tuple such that $q_f \in Q_1 \cap \cdots \cap Q_m$. This contradicts the fact that the run of $\mathcal{T}'$ is accepting.

Conversely, take some accepting run of $\mathcal{T}'$. The accepting state $(Q_1, \ldots, Q_m)$ of this run has no final state $q_f \in Q_1 \cap \cdots \cap Q_m$ and thus there can be no accepting run of $\mathcal{T}$. □

A.3 Membership

In this section we prove the following.

Proposition A.5. *The membership problem for annotated stack tree automata is in linear time.*

Proof. We give an algorithm which checks if a tree t is recognised by an automaton.

We start by labelling every leaf labelled with control p with $\{p\}$.

For every node v such that all its sons have been labelled, we label it by every state q such that there exist transitions $q \leftarrow_{1/m} (q_1, r_1), \cdots, q \leftarrow_{m/m} (q_m, r_m)$ such that each son vi is labelled by a set containing q_i and the stack labelling vi is accepted by r_i. Note, checking the acceptance of a stack from r_i can be done in linear time [5].

If we can label the root by a final state, the tree is accepted (as at each step, if we can label a node by a state, there is a run in which it is labelled by this state), otherwise, it is not.

As knowing if a stack is accepted from a given state is linear in the size of the stack, and we visit each node once, and explore with it once each possible transitions, the complexity of this algorithm is linear in the size of the tree. $\square$

A.4 Emptiness

In this section we prove the following.

Proposition A.6. *The emptiness problem for annotated stack tree automata is in PSPACE-complete.*

Proof. We give the following algorithm:

We set Marked $= \mathbb{P}$.

If there exists a q which is not in Marked such that, there is some m such that for each $i \leq m$ we have $q \leftarrow_{i/m} (q', r')$, with $q' \in$ Marked and there exists a stack recognised from r', we add q to Marked.

We stop when there does not exist such a state.

If Marked $\cap \mathcal{F} = \emptyset$, the recognised language is empty, otherwise, there is at least one tree recognised.

There are at most $|\mathbb{Q}|$ steps in the algorithm, and the complexity of the emptiness problem for the states r is PSPACE. Thus, the algorithm runs in PSPACE. $\square$

A.5 Automata Transformations

In this section we show that annotated stack tree automata can always be transformed to meet the assumptions of the saturation algorithm.

Take a stack tree automaton

$$\mathcal{T} = (\mathbb{Q}, \mathbb{R}_n, \ldots, \mathbb{R}_1, \Sigma, \Delta, \Delta_n, \ldots, \Delta_1, \mathbb{P}, \mathbb{F}, \mathbb{F}_n, \ldots, \mathbb{F}_1) \ .$$

We normalise this automaton as follows. It can be easily seen at each step that we preserve the language accepted by the automaton.

First we ensure that there are no transitions

$$p \leftarrow_{i/m} (q, r) \ .$$

We do this by introducing a new state q_p for each $p \in \mathbb{P}$. Then, we replace each

$$p \leftarrow_{i/m} (q, r)$$

with

$$q_p \leftarrow_{i/m} (q, r)$$

and for each

$$q \leftarrow_{i/m} (p, r)$$

in the resulting automaton, add a transition (not replace)

$$q \leftarrow_{i/m} (q_p, r) \ .$$

Thus, we obtain an automaton with no incoming transitions to any p.

To ensure unique states labelling transitions, we replace each transition

$$q \leftarrow_{i/m} (q', r)$$

with a transition

$$q \leftarrow_{i/m} \left(q', r_{(q,q')}\right)$$

where there is one $r_{(q,q')}$ for each pair of states q, q'. Then when $n > 1$ we have a transition $r_{(q,q')} \xrightarrow{r'} R$ for each $r \xrightarrow{r'} R$. Notice, if there are multiple possible r then $r_{(q,q')} \xrightarrow{r'} R$ accepts the union of their languages. Furthermore, $r_{(q,q')}$ has no incoming transitions. Moreover, we do not remove any transitions from r but observe that r is no longer initial. When $n = 1$ we have a transition $r_{(r,R)} \xrightarrow[R_{\mathrm{br}}]{a} R'$ for each $r \xrightarrow[R_{\mathrm{br}}]{a} R'$.

We then iterate from $k = n$ down to $k = 3$ performing a similar transformation to the above. That is, we replace each transition in the order-k transition set

$$r \xrightarrow{r'} R$$

with a transition

$$r \xrightarrow{r_{(r,R)}} R$$

where there is one $r_{(r,R)}$ for each pair of r and R. Then we have a transition $r_{(r,R)} \xrightarrow{r''} R'$ for each $r' \xrightarrow{r''} R'$. Again, if there are multiple possible r' then $r_{(r,R)} \xrightarrow{r''} R'$ accepts the union of their languages. Furthermore, $r_{(r,R)}$ has no incoming transitions.

Finally, for $k = 2$ the procedure is similar. We replace each transition in the order-2 transition set

$$r \xrightarrow{r'} R$$

with a transition

$$r \xrightarrow{r_{(r,R)}} R$$

where there is one $r_{(r,R)}$ for each pair of r and R. Then we have a transition $r_{(r,R)} \xrightarrow[R_{\mathrm{br}}]{a} R'$ for each $r' \xrightarrow[R_{\mathrm{br}}]{a} R'$.

A.6 Alternative Tree Automaton Definition

An alternative definition of stack tree automata would use transitions

$$q \leftarrow (q_1, r_1), \ldots, (q_m, r_m)$$

instead of

$$q \leftarrow_{1/m} (q_1, r_1), \ldots, q \leftarrow_{m/m} (q_m, r_m) \ .$$

However, due to the dependency such transitions introduce between $r_1, \ldots, r_m$ it is no longer possible to have a unique sequence $r_1, \ldots, r_m$ for each sequence $q, q_1, \ldots, q_m$ (one cannot simply union the candidates for each r_i).

For example suppose we had $q \leftarrow (q_1, r_1), (q_2, r_2)$ and $q \leftarrow (q_1, r_1'), (q_2, r_2')$ where r_1 accepts s_1, r_1' accepts s_1', r_2 accepts s_2, and r_2' accepts s_2'. If we were to replace these two transitions with $q \leftarrow (q_1, r_1 \cup r_1'), (q_2, r_2 \cup r_2')$ we would mix up the two transitions, allowing, for example, the first child to be labelled by s_1 and the second by s_2'.

At a first glance, our tree automaton model may appear weaker since we cannot enforce dependencies between the candidate r_is in

$$q \leftarrow_{1/m} (q_1, r_1), \ldots, q \leftarrow_{m/m} (q_m, r_m) \ .$$

However, it turns out that we can overcome this problem with new copies of q.

That is, suppose we had a set Δ of transitions of the form

$$q \leftarrow (q_1, r_1), \ldots, (q_m, r_m) \ .$$

We could simulate the resulting tree automaton using our model by introducing a state (q, δ) for each q and δ.

Given a transition δ of the above form, we can use a family of rules

$$(q, \delta) \leftarrow_{1/m} ((q_1, \delta_1), r_1), \ldots, (q, \delta) \leftarrow_{m/m} ((q_m, \delta_m), r_m)$$

for all sequences $\delta_1, \ldots, \delta_m$ of Δ. (Note that, although there are an exponential number of such families, we can create them all from a polynomial number of transitions). Note that when $q_i = p$ we would use p on the right hand side instead of (q_i, δ_i) (recalling that p has no incoming transitions).

B Completeness of Saturation

Lemma B.1 (Completeness of Saturation)**.** *The automaton $\mathcal{T}$ obtained by saturation from $\mathcal{T}_0$ is such that* $\mathrm{Pre}^*_{\mathcal{G}}(\mathcal{T}_0) \subseteq \mathcal{L}(\mathcal{T})$.

Proof. Completeness is proved via a straightforward induction over the length of the run witnessing $t \in \mathrm{Pre}^*_{\mathcal{G}}(\mathcal{T}_0)$. In the base case we have $t \in \mathcal{L}(\mathcal{T}_0)$ and since $\mathcal{T}$ was obtained only by adding transitions to $\mathcal{T}_0$, we are done.

For the induction, take $t \in \theta(t')$ where $t' \in \mathrm{Pre}^*_{\mathcal{G}}(\mathcal{T}_0)$ and by induction $\mathcal{T}$ has an accepting run of t'. We show how the transitions added by saturation can be used to build from the run over t' an accepting run over t.

We first consider the cases where θ adds or removes nodes to/from the tree. The remaining cases when the stack contents are altered are almost identical to the ICALP 2012 proof, and hence are left until the end for the interested reader.

- When $\theta = p \xrightarrow{+} (p_1, \ldots, p_m)$ was applied to node $t_{\bullet_j}$ of t, we have

$$t' = t\big[t_{\bullet_j} \to s\big]\big[t_{\bullet_j}1 \to (p_1, s)\big] \cdots \big[t_{\bullet_j}m \to (p_m, s)\big]$$

  where (p, s) labelled $t_{\bullet_j}$.

  Take the initial transitions over $t_{\bullet_j}$ and $t_{\bullet_j}1$ to $t_{\bullet_j}m$ of the accepting run of t'

$$q \leftarrow_{i/m'} (q_1, a, R_{\text{br}}, R_1, \ldots, R_n)$$

  and

$$q_1 \leftarrow_{1/m} \left(p_1, a, R^1_{\text{br}}, R^1_1, \ldots, R^1_n\right), \ldots, q_1 \leftarrow_{m/m} (p_m, a, R^m_{\text{br}}, R^m_1, \ldots, R^m_n)$$

  where the components of s were accepted from R_{br}, $R_1, \ldots, R_n$ and R^1_{br}, $R^1_1, \ldots, R^1_n$, $\ldots$, R^m_{br}, $R^m_1, \ldots, R^m_n$.

  By saturation we also have

$$q \leftarrow_{i/m'} (p, a, R'_{\text{br}}, R'_1, \ldots, R'_n)$$

  where $R'_{\text{br}} = R_{\text{br}} \cup R^1_{\text{br}} \cup \cdots \cup R^m_{\text{br}}$ and for all k, we have $R'_k = R_1 \cup R^1_k \cup \cdots \cup R^m_k$ from which we obtain a run of $\mathcal{T}$ over t by simply replacing the transitions of the run over t' identified above with δ.

- When $\theta = (p_1, \ldots, p_m) \xrightarrow{-} p$ was applied to nodes $t_{\bullet_j}$ to $t_{\bullet_{j+m-1}}$ of t, we have $t' = t \backslash \left\{t_{\bullet_j}, \ldots, t_{\bullet_{j+m-1}}\right\}$ and $t_{\bullet_j}$, $\ldots$, $t_{\bullet_{j+m}}$ were the only children of their parent v. Moreover, let (p_1, s_1) label $t_{\bullet_j}$, and $\ldots$ and, (p_m, s_m) label $t_{\bullet_{j+m-1}}$ and v have the stack s in t and (p, s) label v in t'.

  The initial transition over v of the accepting run of t' was from state p By saturation we have

$$\delta_1 = p \leftarrow_{1/m} (p_1, a_1, \emptyset, \emptyset, \ldots, \emptyset), \quad \ldots, \quad \delta_m = p \leftarrow_{m/m} (p_m, a_m, \emptyset, \emptyset, \ldots, \emptyset)$$

  for the $a_1, \ldots, a_m$ at the top of s_1, $\ldots$, s_m respectively. We get from this a run of $\mathcal{T}$ over t by adding δ_1 to δ_m to the run over t' to read the nodes $t_{\bullet_j}$ to $t_{\bullet_{j+m-1}}$.

We now consider the cases where θ applies a stack operation to a single node ${t'}_{\bullet_j}$ of t'. Let

$$\delta' = q \leftarrow_{i/m} (p', a, R_{\text{br}}, R_1, \ldots, R_m)$$

be the transition applied at node ${t'}_{\bullet_j}$ in the run. Additionally, let s' be the stack labelling the node, and p' be the control state.

There is a case for each type of stack operation, all of which are almost identical to the ICALP 2012 proof. In all cases below, t has the same tree structure as t' and only differs on the labelling of ${t'}_{\bullet_j} = t_{\bullet_j}$.

- When $\theta = p \xrightarrow{\text{rew}_{b \to a}} p'$ then we also added the transition

$$\delta = q \leftarrow_{i/m} (p, b, R_{\text{br}}, R_1, \ldots, R_m)$$

  to $\mathcal{T}$. We have

$$s' = a^{s_{\text{br}}} :_1 s_1 :_2 \cdots :_n s_n$$

  and since $t_{\bullet_j}$ is labelled by p and the stack

$$s = b^{s_{\text{br}}} :_1 s_1 :_2 \cdots :_n s_n$$

  we obtain an accepting run of t by simply replacing the application of δ' with δ.

- When $\theta = p \xrightarrow{\text{push}_1^k} p'$ then when $k > 1$ we have

  $$s' = a^{s_k} :_1 a^{s_{\text{br}}} s_1 :_2 \cdots :_n s_n \ .$$

  Let

  $$R_1 \xrightarrow[R'_{\text{br}}]{a} (R'_1)$$

  be the first transitions used to accept $a^{s_{\text{br}}}$. From the saturation algorithm we also added

  $$\delta = q \leftarrow_{i/m} (p, a, R'_{\text{br}}, R'_1, R_2, \ldots, R_{k-1}, R_k \cup R_{\text{br}}, R_{k+1}, \ldots, R_n)$$

  to $\mathcal{T}$. Since $t_{\bullet_j}$ is labelled by p and the stack

  $$s = a^{s_{\text{br}}} :_1 s_1 :_2 \cdots :_n s_n$$

  we obtain an accepting run of t by replacing the application of δ' with δ. This follows because s'_1 was accepted from R'_1, s_{br} from R'_{br} and s_k was accepted from both R_k and R_{br}.

  When $k = 1$ we have

  $$s' = a^{s_1} :_1 a^{s_{\text{br}}} s_1 :_2 \cdots :_n s_n \ .$$

  Let

  $$R_1 \xrightarrow[R'_{\text{br}}]{a} (R'_1)$$

  be the first transitions used to accept $a^{s_{\text{br}}}$. From the saturation algorithm we also added

  $$\delta = q \leftarrow_{i/m} (p, a, R'_{\text{br}}, R'_1 \cup R_{\text{br}}, R_2, \ldots, R_n)$$

  to $\mathcal{T}$. Since $t_{\bullet_j}$ is labelled by p and the stack

  $$s = a^{s_{\text{br}}} :_1 s_1 :_2 \cdots :_n s_n$$

  we obtain an accepting run of t by replacing the application of δ' with δ. This follows because s'_1 was accepted from R'_1, s_{br} from R'_{br} and s_k was accepted from both R_k and R_{br}.

- When $\theta = p \xrightarrow{\text{push}_k} p'$ then we have

  $$s' = s_k :_k s_k :_{k+1} s_{k+1} \cdots :_n s_n \quad \text{and} \quad s_k = a^{s_{\text{br}}} :_1 s'_1 :_2 \cdots :_{(k-1)} s_{k-1} \ .$$

  Let

  $$R_k \xrightarrow[R'_{\text{br}}]{a} (R'_1, \ldots, R'_k)$$

  be the transitions use to accept the first character of the second appearance of s_k. From the saturation algorithm we also added $\delta =$

  $$q \leftarrow_{i/m} \left(p, a, R_{\text{br}} \cup R'_{\text{br}}, R_1 \cup R'_1, R_2 \cup R'_2, \ldots, R_{k-1} \cup R'_{k-1}, R'_k, R_{k+1}, \ldots, R_n\right)$$

  to $\mathcal{T}$. Since $t_{\bullet_j}$ is labelled by p and the stack

  $$s = a^{s_{\text{br}}} :_1 s_1 :_2 \cdots :_n s_n$$

  we obtain an accepting run of t by replacing the application of δ' with δ. This follows because stacks s_1 to s_{k-1} are accepted from R_1 and R'_1 to R_{k-1} and R'_{k-1} respectively, s_{br} from R_{br} and R'_{br}, and the remainder of the stack from R'_k, R_{k+1}, $\ldots$, R_n.

- When $\theta = p \xrightarrow{\mathrm{pop}_k} p'$ Then we have

  $$s' = s_k :_{k+1} s_{k+1} \cdots :_n s_n$$

  and

  $$s = a^{s_{\mathrm{br}}} :_1 s_1 :_2 \cdots :_n s_n$$

  for some a, s_{br}, s_1, ..., s_{k-1}. We break down δ' to find r_k such that

  $$q \leftarrow_{i/m} (p', r_k, R_{k+1}, \ldots R_n)$$

  where r_k accepts s_k and R_{k+1} through to R_n accept s_{k+1} through to s_n respectively. By saturation we added the transition

  $$\delta = q \leftarrow_{i/m} (p, a, \emptyset, \emptyset, \ldots, \emptyset, \{r_k\}, R_{k+1}, \ldots, R_n)$$

  from which we obtain an accepting run of s with p as required.

- When $\theta = p \xrightarrow{\mathrm{collapse}_k} p'$ Then we have

  $$s' = s_{\mathrm{br}}, :_{k+1} s_{k+1} \cdots :_n s_n$$

  and

  $$s = a^{s_{\mathrm{br}}} :_1 s_1 :_2 \cdots :_n s_n$$

  for some a, s_{br}, s_1, ..., s_k. We break down δ' to find r_{br} such that

  $$q \leftarrow_{i/m} (p', r_{\mathrm{br}}, R_{k+1}, \ldots R_n)$$

  where r_k accepts s_{br} and R_{k+1} through to R_n accept s_{k+1} through to s_n respectively. By saturation we added the transition

  $$\delta = q \leftarrow_{i/m} (p, a, \{r_{\mathrm{br}}\}, \emptyset, \ldots, \emptyset, R_{k+1}, \ldots, R_n)$$

  from which we obtain an accepting run of s with p as required.

Thus, in all cases we find an accepting run of $\mathcal{T}$, which completes the proof. □

C Soundness of Saturation

We prove that the automaton $\mathcal{T}$ constructed by saturation only accepts trees in $\mathrm{Pre}^*_{\mathcal{G}}(\mathcal{T}_0)$. The proof relies on the notion of a "sound" automaton. There are several stages to the proof.

- We assign meanings to each state of the automaton that ultimately capture inclusion in $\mathrm{Pre}^*_{\mathcal{G}}(\mathcal{T}_0)$.
- We use these meanings to derive a notion of sound transitions.
- We define a sound automaton based on the notion of sound transitions.
- We show sound tree automata only accept trees in $\mathrm{Pre}^*_{\mathcal{G}}(\mathcal{T}_0)$.
- We show the initial automaton $\mathcal{T}_0$ is sound, and moreover, each saturation step preserves soundness, from which we conclude soundness of the saturation algorithm.

To define the meanings of the states we need to reason about partial runs of our stack tree automata. Hence for a tree automaton $\mathcal{T}$ we define

$$\mathcal{L}_W(\mathcal{T})$$

to accept trees over the set of control states $\mathbb{Q}$ (instead of $\mathbb{P}$). That is, we can accept prefixes of trees accepted by $\mathcal{T}$ by labelling the leaves with the states that would have appeared on an accepting run of the full tree.

Furthermore, we write

$$\mathcal{L}_{q_1,\ldots,q_m}(\mathcal{T})$$

to denote the set of trees t in $\mathcal{L}_W(\mathcal{T})$ such that t has m leaves and the "control" states (which now includes all states in $\mathbb{Q}$) appearing on the leaves are $q_1, \ldots, q_m$ respectively. As a special case, $\mathcal{L}_{q_f}(\mathcal{T})$ for all $q_f \in \mathbb{F}$ contains only the empty tree.

C.1 Meaning of a State

We assign to each state of the automaton a "meaning". This meaning captures the requirement that the states p of the automaton should accept $\mathrm{Pre}^*_{\mathcal{G}}(\mathcal{T}_0)$, while the meanings of the non-initial states are given by the automaton itself (i.e. the states should accept everything they accept). For states accepting stacks, the non-initial states again have the trivial meaning (they should accept what they accept), while the meanings of the initial states are inherited from the transitions that they label.

We write $\tilde{q}$ to denote a sequence $q_1, \ldots, q_m$ and $|q_1, \ldots, q_m|$ is m.

Let $\mathcal{V}$ be a partial mapping of nodes to states in $\mathbb{Q}$, let $\emptyset$ be the empty mapping, and let

$$\mathcal{V}[v \to q](v') = \begin{cases} q & v = v' \\ \mathcal{V}(v') & v \neq v' \end{cases} .$$

We use these mappings in definition below to place conditions on nodes in the tree that restrict runs witnessing membership in $\mathrm{Pre}^*_{\mathcal{G}}(\mathcal{T}_0)$.

Definition C.1 ($t \models_{\mathcal{V}} q_1, \ldots, q_m$)**.** *If t has m leaves labelled $q_1, \ldots, q_m$ respectively then $t \models_{\mathcal{V}} q_1, \ldots, q_m$ whenever $t \in \mathrm{Pre}^*_{\mathcal{G}}(\mathcal{L}_W(\mathcal{T}_0))$ and there is a run to some $t' \in \mathcal{L}_W(\mathcal{T}_0)$ such that – fixing an accepting run of $\mathcal{T}_0$ over t' – for all nodes v of t with $\mathcal{V}(v) = q$, then*

- *if $q \in \mathbb{P}$ then v appears as a leaf during the run and on the first such tree in the run, v has control state q.*
- *if $q \notin \mathbb{P}$ then v is not a leaf of any tree on the run and the accepting run of $\mathcal{T}$ over t' labels v with q.*

As a special case, when t is empty we have $t \models_{\emptyset} q_f$ and $q_f \in \mathbb{F}$.

Once we have assigned meanings to the states of $\mathbb{Q}$, we need to derive meanings for the states in $\mathbb{R}_n, \ldots, \mathbb{R}_1$. We first introduce some notation.

$$t +_i (q_1, s_1), \ldots, (q_m, s_m) = t[t_{\bullet_i} \to s][t_{\bullet_i}1 \to (q_1, s_1)] \cdots [t_{\bullet_i}m \to (q_m, s_m)]$$

when t is non-empty and s is the stack labelling $t_{\bullet_i}$ in t. When t is empty we have

$$t +_0 (q_1, s_1)$$

is the single-node tree labelled by (q_1, s_1).

In the definition below we assign meanings to states accepting stacks. The first case is the simple case where a state is non-initial, and its meaning is to accept the set of stacks it accepts.

The second case derives a meaning of a state in $\mathbb{R}_k$ by inheriting the meaning from the states of $\mathbb{R}_{k+1}$. Intuitively, if we have a transition $r_{k+1} \xrightarrow{r_k} R_{k+1}$ then the meaning of r_k is that it should accept all stacks that could appear on top of a stack in the meaning of R_{k+1} to form a stack in the meaning of r_k.

The final case is a generalisation of the above case to trees. The states in $\mathbb{R}_n$ should accept all stacks that could appear on a node of the tree consistent with a run of the stack tree automaton and the meanings of the states in $\mathbb{Q}$.

Definition C.2 ($s \models r$)**.** *For any $R \subseteq \mathbb{R}_k$ and any order-k stack s, we write $s \models R$ if $s \models r$ for all $r \in R$. We define $s \models r$ by a case distinction on r.*

1. *When r is a non-initial state in $\mathbb{R}_k$, then we have $s \models r$ if s is accepted from r.*
2. *If r_k is an initial state in $\mathbb{R}_k$ with $k < n$ labelling a transition $r_{k+1} \xrightarrow{r_k} R_{k+1} \in \Delta_{k+1}$ then we have $s \models r_k$ if for all stacks s' such that $s' \models R_{k+1}$ we have $s :_{k+1} s' \models r_{k+1}$.*
3. *We have $s \models r$ where $q \leftarrow_{i/m} (q', r)$ if for all transitions*

$$q \leftarrow_{1/m} (q_1, r_1), \ldots, q \leftarrow_{m/m} (q_m, r_m)$$

trees $t \models_{\mathcal{V}} \tilde{q}_1, q, \tilde{q}_2$ and stacks $s_1, \ldots, s_m$ such that

$$t +_j (q_1, s_1), \ldots, (q_m, s_m) \models_{\mathcal{V}[t_{\bullet_j} \to q]} \tilde{q}_1, q_1, \ldots, q_m, \tilde{q}_2$$

where $j = |\tilde{q}_1| + 1$, we have

$$t +_j (q_1, s_1), \ldots, (q_{i-1}, s_{i-1}), (q', s), (q_{i+1}, s_{i+1}), \ldots, (q_m, s_m) \\ \models_{\mathcal{V}[t_{\bullet_j} \to q]} \tilde{q}_1, q_1, \ldots, q_{i-1}, q', q_{i+1}, \ldots, q_m, \tilde{q}_2 \ .$$

Note that item 3 of the definition of $\models$ contains a vacuity in that there may be no $s_1, \ldots, s_m$ satisfying the antecedent (in which case all stacks would be in the meaning of r). Hence, we require a non-redundancy condition on the automata.

Definition C.3 (Non-Redundancy)**.** *An order-n annotated stack tree automaton*

$$\mathcal{T} = (\mathbb{Q}, \mathbb{R}_n, \ldots, \mathbb{R}_1, \Sigma, \Delta, \Delta_n, \ldots, \Delta_1, \mathbb{P}, \mathbb{F}, \mathbb{F}_n, \ldots, \mathbb{F}_1)$$

is non-redundant *if for all $q \in \mathbb{Q}$ we have that either q has no-incoming transitions, or there exist*

$$q \leftarrow_{1/m} (q_1, r_1), \ldots, q \leftarrow_{m/m} (q_m, r_m) \in \Delta$$

such that for all $t \models_{\mathcal{V}} \tilde{q}_1, q, \tilde{q}_2$ there exist $s_1, \ldots, s_m$ such that

$$t +_j (q_1, s_1), \ldots, (q_m, s_m) \models_{\mathcal{V}[t_{\bullet_j} \to q]} \tilde{q}_1, q_1, \ldots, q_m, \tilde{q}_2$$

where $j = |\tilde{q}_1| + 1$.

This property can be easily satisfied in $\mathcal{T}_0$ by removing states q that do not satisfy the non-redundancy conditions (this does not change the language since there were no trees that could be accepted using q). We show later that the property is maintained by saturation.

C.2 Soundness of a Transition

After assigning meanings to states, we can define a notion of soundness for the transitions of the automata. Intuitively, a transition is sound if it respects the meanings of its source and target states.

One may derive some more intuition by considering a transition $q \xrightarrow{a} q'$ of a finite word automaton. The transition would be sound if, for every word w in the meaning of q', the same word with an a in front is in the meaning of q. That is, the transition is sound if an a can appear on anything accepted from q'. The following definition translates the same idea to the case of stack trees.

Definition C.4 (Soundness of transitions)**.** *There are two cases given below.*

1. *A transition $r_k \xrightarrow[R_{\mathrm{br}}]{a} (R_1, \ldots, R_k)$ is sound if for any $s_1 \models R_1$, ..., $s_k \models R_k$ and $s_{\mathrm{br}} \models R_{\mathrm{br}}$ we have $a^{s_{\mathrm{br}}} :_1 s_1 :_2 \cdots :_k s_k \models r_k$.*
2. *A transition*
$$q \leftarrow_{i/m} (q', a, R_{\mathrm{br}}, R_1, \ldots, R_n),$$
is sound if for all trees $t \models_{\mathcal{V}} \tilde{q}_1, q, \tilde{q}_2$ and stacks $s_1 \models R_1$, $\ldots s_m \models R_m$, and $s_{\mathrm{br}} \models R_{\mathrm{br}}$ and for all
$$q \leftarrow_{1/m} (q_1, r_1), \ldots, q \leftarrow_{m/m} (q_m, r_m)$$
and stacks $s'_1, \ldots, s'_m$ such that
$$t +_j (q_1, s'_1), \ldots, (q_m, s'_m) \models_{\mathcal{V}[t_{\bullet_j} \to q]} \tilde{q}_1, q_1, \ldots, q_m, \tilde{q}_2$$
where $j = |\tilde{q}_1| + 1$, we have
$$t +_j (q_1, s'_1), \ldots, \left(q_{i-1}, s'_{i-1}\right), (q', s), \left(q_{i+1}, s'_{i+1}\right), \ldots, (q_m, s'_m) \models_{\mathcal{V}[t_{\bullet_j} \to q]} \tilde{q}_1, q_1, \ldots, q_{i-1}, q', q_{i+1}, \ldots, q_m, \tilde{q}_2 \ .$$
where
$$s = a^{s_{\mathrm{br}}} :_1 s_1 :_2 \cdots :_n s_n \ .$$

In the proof, we will have to show that saturation builds a sound automaton. This means proving soundness for each new transition. The following lemma shows that it suffices to only show soundness for the outer collections of transitions.

Lemma C.1 (Cascading Soundness)**.** *If a transition*
$$q \leftarrow_{i/m} (q', a, R_{\mathrm{br}}, R_1, \ldots, R_n),$$
is sound then all transitions $r_k \xrightarrow[R_{\mathrm{br}}]{a} (R_1, \ldots, R_k)$ appearing within the transition are also sound.

Proof. We march by induction. Initially $k = n$ and we have $r \xrightarrow[R_{\mathrm{br}}]{a} (R_1, \ldots, R_n)$ where $q \leftarrow_{i/m} (q_i, r)$. To prove soundness of the transition from r, take $s_1 \models R^i_1$, ..., $s_n \models R^i_n$, and $s_{\mathrm{br}} \models R^i_{\mathrm{br}}$. We need to show
$$s = a^{s_{\mathrm{br}}} :_1 s_1 :_2 \cdots :_n s_n \models r \ .$$

This is the case if, letting $j = |\tilde{q}_1|$, for all transitions

$$q \leftarrow_{1/m} (q_1, r_1), \ldots, q \leftarrow_{m/m} (q_m, r_m)$$

trees $t \models_{\mathcal{V}} \tilde{q}_1, q, \tilde{q}_2$ and stacks $s_1, \ldots, s_m$ such that

$$t +_j (q_1, s_1), \ldots, (q_m, s_m) \models_{\mathcal{V}[t_{\bullet_j} \to q]} \tilde{q}_1, q_1, \ldots, q_m, \tilde{q}_2$$

we have

$$t +_j (q_1, s_1), \ldots, (q_{i-1}, s_{i-1}), (q', s), (q_{i+1}, s_{i+1}), \ldots, (q_m, s_m) \\ \models_{\mathcal{V}[t_{\bullet_j} \to q]} \tilde{q}_1, q_1, \ldots, q_{i-1}, q', q_{i+1}, \ldots, q_m, \tilde{q}_2 \ .$$

These properties are derived immediately from the fact that

$$q \leftarrow_{i/m} (q', a, R_{\mathrm{br}}, R_1, \ldots, R_n),$$

is sound, hence we are done.

When $k < n$ we assume $r_{k+1} \xrightarrow[R_{\mathrm{br}}]{a} (R_1, \ldots, R_{k+1})$ is sound and $r_{k+1} \xrightarrow{r_k} R_{k+1}$. We show $r_k \xrightarrow[R_{\mathrm{br}}]{a} (R_1, \ldots, R_k)$ is also sound. For this, we take any stacks $s_1 \models R_1$, $\ldots s_k \models R_k$, and $s_{\mathrm{br}} \models R_{\mathrm{br}}$. We need to show

$$s = a^{s_{\mathrm{br}}} :_1 s_1 :_2 \cdots :_k s_k \models r_k \ .$$

For this, we need for all $s' \models R_{k+1}$ that $s :_{(k+1)} s' \models r_{k+1}$. From the soundness of $r_{k+1} \xrightarrow[R_{\mathrm{br}}]{a} (R_1, \ldots, R_{k+1})$ we have

$$s :_{(k+1)} s' = a^{s_{\mathrm{br}}} :_1 s_1 :_2 \cdots :_{(k+1)} s_{k+1} \models r_{k+1}$$

and we are done. □

C.3 Soundness of Annotated Stack Tree Automata

We will prove the saturation constructs a sound automaton. We first define what it means for an automaton to be sound and prove that a sound automaton only accepts trees in $\mathrm{Pre}^*_{\mathcal{G}}(\mathcal{T}_0)$.

Definition C.5 (Soundness of Annotated Stack Tree Automata)**.** *An annotated stack tree automaton $\mathcal{T}$ is sound if*

1. *$\mathcal{T}$ is obtained from $\mathcal{T}_0$ by adding new initial states to $\mathbb{R}_1, \ldots, \mathbb{R}_n$ and transitions starting at initial states, and*
2. *in $\mathcal{T}$, all transitions*

   $$q \leftarrow_{i/m} (q', a, R_{\mathrm{br}}, R_1, \ldots, R_n)$$

   and

   $$r_k \xrightarrow[R_{\mathrm{br}}]{a} (R_1, \ldots, R_k)$$

   are sound, and
3. *$\mathcal{T}$ is non-redundant.*

We show that a sound annotated stack tree automaton can only accept trees belonging to $\mathrm{Pre}^*_{\mathcal{G}}(\mathcal{T}_0)$. In fact, we prove a more general result. In the following lemma, note the particular case where $t \in \mathcal{L}_{\tilde{q}}(\mathcal{T})$ and $\tilde{q}$ is a sequence of states in $\mathbb{P}$ then we have $t \in \mathrm{Pre}^*_{\mathcal{G}}(\mathcal{T}_0)$. That is, $\mathcal{L}(\mathcal{T}) \subseteq \mathrm{Pre}^*_{\mathcal{G}}(\mathcal{T}_0)$.

Lemma C.2 (Sound Acceptance)**.** *Let $\mathcal{T}$ be a sound annotated stack automaton. For all $t \in \mathcal{L}_{\tilde{q}}(\mathcal{T})$ we have $t \models_{\emptyset} \tilde{q}$.*

Before we can prove the result about trees, we first prove a related result about stacks. This result and proof is taken almost directly from ICALP 2012 [5].

Lemma C.3 (Sound Acceptance of Stacks)**.** *Let $\mathcal{T}$ be a sound annotated stack automaton. If $\mathcal{T}$ accepts an order-k stack s from $r \in \mathbb{R}_k$ then $s \models r$.*

Proof. We proceed by induction on the size of the stack (where the size of an annotated stack is defined to be the size of a tree representing the stack).

Let s be an order-k stack accepted from a state $r \in \mathbb{R}_k$. We assume that the property holds for any smaller stack.

If s is empty then r is a final state. Recall that by assumption final states are not initial, hence r is not initial. It follows that the empty stack is accepted from r in $\mathcal{T}_0$ and hence $s \models r$.

If s is a non-empty stack of order-1, then $s = a^{s_{\mathrm{br}}} :_1 s_1$. As s is accepted from r, there exists a transition $r \xrightarrow[R_{\mathrm{br}}]{a} (R_1)$ such that s_1 is accepted from R_1 and s_{br} is accepted from R_{br}. By induction we have $s_1 \models R_1$ and $s_{\mathrm{br}} \models R_{\mathrm{br}}$. Since the transition is sound, we have $s \models r$.

If s is a non-empty stack of order-k, then $s = s_{k-1} :_k s_k$. As s is accepted from r, there exists a transition $r \xrightarrow{r'} R$ such that s_k is accepted from R and s_{k-1} is accepted from r'. By induction we have $s_{k-1} \models r'$ and $s_k \models R_k$. Thus, by the definition of $s_{k-1} \models r'$ we also have $s = s_{k-1} :_k s_k \models r$. □

We are now ready to prove Lemma C.2 (Sound Acceptance).

Proof of Lemma C.2 (Sound Acceptance). We proceed by induction on the number of nodes in the tree. In the base case, we have $t \in \mathcal{L}_{q_f}(\mathcal{T})$ for some $q_f \in \mathbb{F}$ and t is empty. Thus, we immediately have $t \models_{\emptyset} q_f$.

Thus, take some non-empty $t \in \mathcal{L}_{\tilde{q}}(\mathcal{T})$. Let the sequence $t_{\bullet_i}, \ldots, t_{\bullet_{i+m}}$ be the first complete group of siblings that are all leaf nodes and let $\tilde{q} = \tilde{q}_1, q_1, \ldots, q_m, \tilde{q}_2$ be the decomposition of $\tilde{q}$ such that $\tilde{q}_1$ is of length $(i-1)$. That is, $q_1, \ldots, q_m$ label the identified leaves of t. Furthermore, let $s_1, \ldots, s_m$ be the respective stacks labelling these leaves. Take the set of transitions

$$q \leftarrow_{1/m} (q_1, r_1), \ldots q \leftarrow_{m/m} (q_m, r_m)$$

that are used in the accepting run of t and the identified leaves. Let t' be the tree obtained by removing $t_{\bullet_i}, \ldots, t_{\bullet_{i+m'}}$. We have $t' \in \mathcal{L}_{\tilde{q}_1, q, \tilde{q}_2}(\mathcal{T})$ and by induction $t' \models_{\emptyset} \tilde{q}_1, q, \tilde{q}_2$.

Since q has incoming transitions and $\mathcal{T}$ is non-redundant, we know there exists

$$q \leftarrow_{1/m} (q'_1, r'_1), \ldots q \leftarrow_{m/m} (q'_m, r'_m)$$

and $s'_1, \ldots, s'_m$ such that

$$t' +_i (q'_1, s'_1), \ldots, (q'_m, s'_m) \models_{\emptyset[t_{\bullet_i} \to q]} \tilde{q}_1, q'_1, \ldots, q'_m, \tilde{q}_2 \ .$$

Since $s_1 \models r_1$ we infer from the definition of $\models$ at r_1 that

$$t' +_i (q_1, s_1), (q_2', s_2'), \ldots, (q_m', s_m') \models_{\emptyset[t_{\bullet_i} \to q]} \tilde{q}_1, q_1, q_2', \ldots, q_m', \tilde{q}_2 \ .$$

By repeated applications of the above for each $1 < j \leq m$, we obtain

$$t' +_i (q_1, s_1), (q_2, s_2), \ldots, (q_m, s_m) \models_{\emptyset[t_{\bullet_i} \to q]} \tilde{q}_1, q_1, \ldots, q_m, \tilde{q}_2 \ .$$

This implies $t \models_{\emptyset} \tilde{q}$ since $\models_{\emptyset}$ is less restrictive than $\models_{\emptyset[t_{\bullet_i} \to q]}$. □

C.4 Soundness of Saturation

We first prove that $\mathcal{T}_0$ is sound, and then that saturation maintains the property.

Lemma C.4 (Soundness of $\mathcal{T}_0$)**.** *The initial automaton $\mathcal{T}_0$ is sound.*

Proof. It is trivial that $\mathcal{T}_0$ is obtained from $\mathcal{T}_0$, and moreover, we assume the non-redundancy condition. Hence, From Lemma C.1 (Cascading Soundness) we only need to prove soundness of non-initial transitions of the form

$$r_k \xrightarrow[R_{\mathrm{br}}]{a} (R_1, \ldots, R_n)$$

and for transitions in Δ.

We first show the case for non-initial

$$r_k \xrightarrow[R_{\mathrm{br}}]{a} (R_1, \ldots, R_n)$$

which is the same as in ICALP 2012. First note that $R_1, \ldots, R_n$ and R_{br} do not contain initial states. Then we take $s_1 \models R_1, \ldots s_k \models R_k$ and $s_{\mathrm{br}} \models R_{\mathrm{br}}$. We have to show $a^{s_{\mathrm{br}}} :_1 s_1 :_2 \cdots :_k s_k \models r_k$. In particular, since r_k is not initial, we only need to construct an accepting run. Since R_i and R_{br} are not initial, we have accepting runs from these states. Hence, we build immediately the run beginning with $r_k \xrightarrow[R_{\mathrm{br}}]{a} (R_1, \ldots, R_n)$.

We now prove the case for

$$q \leftarrow_{i/m} (q', a, R_{\mathrm{br}}, R_1, \ldots, R_n) \ .$$

Thus, take any $s_1 \models R_1, \ldots s_m \models R_m$, and $s_{\mathrm{br}} \models R_{\mathrm{br}}$ and any tree $t \models_{\mathcal{V}} \tilde{q}_1, q, \tilde{q}_2$ and, letting $j = |\tilde{q}_1| + 1$, any

$$q \leftarrow_{1/m} (q_1, r_1), \ldots q \leftarrow_{m/m} (q_m, r_m)$$

and any $s_1', \ldots, s_m'$ such that $t +_j (q_1, s_1') \ldots (q_m, s_m') \models_{\mathcal{V}[t_{\bullet_j} \to q]} \tilde{q}_1, q_1, \ldots, q_m, \tilde{q}_m$. Since initial states have no incoming transitions, we know q is not a control sate. We thus have a run ρ from $t +_j (q_1, s_1') \ldots (q_m, s_m')$ to some $t' \in \mathcal{L}_W(\mathcal{T})$ such that $t_{\bullet_j}$ does not appear as a leaf of any tree in the run.

To prove soundness we argue that

$$t +_j (q_1, s_1'), \ldots, (q_{i-1}, s_{i-1}'), (q', s), (q_{i+1}, s_{i+1}'), \ldots, (q_m, s_m') \\ \models_{\mathcal{V}[t_{\bullet_j} \to q]} \tilde{q}_1, q_1, \ldots, q_{i-1}, q', q_{i+1}, \ldots, q_m, \tilde{q}_2 \quad (1)$$

where $s = a^{s_{\mathrm{br}}} :_1 s_1 :_2 \cdots :_n s_n$. To do so, we take the run ρ obtained above and build a run ρ' by removing all operations applied to nodes that are descendants of $t_{\bullet_j} i$. Observe that ρ' can be applied to

$$t +_j (q_1, s'_1), \ldots, (q_{i-1}, s'_{i-1}), (q', s), (q_{i+1}, s'_{i+1}), \ldots, (q_m, s'_m)$$

since none of the operations apply to a descendant of $t_{\bullet_j} i$. By applying this run we obtain a tree t'' which is t' less all nodes that are strict descendants of $t_{\bullet_j} i$ and where $t_{\bullet_j} i$ is labelled by (q', s). Thus, we take the accepting run of t' witnessing $t' \in \mathcal{L}_W(\mathcal{T}_0)$, remove all nodes that are strict descendants of $t_{\bullet_j} i$ and label $t_{\bullet_j} i$ by q'. This gives us a run witnessing $t'' \in \mathcal{L}_W(\mathcal{T}_0)$ by using

$$q \leftarrow_{i/m} (q', a, R_{\mathrm{br}}, R_1, \ldots, R_n) \ .$$

at $t_{\bullet_j} i$ and the accepting runs from the non-initial R_{br}, $R_1, \ldots, R_n$. This gives us (1) as required. $\square$

We now show that, at every stage of saturation, we maintain a sound automaton.

Lemma C.5 (Soundness of the Saturation Step)**.** *Given a sound automaton $\mathcal{T}$, we have $\mathcal{T}' = \mathcal{F}(\mathcal{T})$ is sound.*

Proof. We analyse all new transitions

$$q \leftarrow_{i/m} (p, a, R^{\mathrm{new}}_{\mathrm{br}}, R^{\mathrm{new}}_1, \ldots, R^{\mathrm{new}}_n) \ .$$

Proving these transitions are sound and do not cause redundancy is sufficient via Lemma C.1 (Cascading Soundness).

Let us begin with the transitions introduced by rules that do not remove nodes from the tree. We argue that for all trees $t \models_{\mathcal{V}} \tilde{q}_1, q, \tilde{q}_2$ and stacks $s_1 \models R^{\mathrm{new}}_1$, $\ldots s_m \models R^{\mathrm{new}}_m$, and $s_{\mathrm{br}} \models R^{\mathrm{new}}_{\mathrm{br}}$ and for all

$$q \leftarrow_{1/m} (q_1, r_1), \ldots, q \leftarrow_{m/m} (q_m, r_m)$$

and stacks $s'_1, \ldots, s'_m$ such that

$$t +_j (q_1, s'_1), \ldots, (q_m, s'_m) \models_{\mathcal{V}[t_{\bullet_j} \to q]} \tilde{q}_1, q_1, \ldots, q_m, \tilde{q}_2$$

where $j = |\tilde{q}_1| + 1$ we have, letting

$$t_1 = t +_j (q_1, s'_1), \ldots, (q_{i-1}, s'_{i-1}), (p, s), (q_{i+1}, s'_{i+1}), \ldots, (q_m, s'_m)$$

and $\tilde{q}'_1 = \tilde{q}_1, q_1, \ldots, q_{i-1}$ and $\tilde{q}'_2 = q_{i+1}, \ldots, q_m, \tilde{q}_2$ that

$$t_1 \models_{\mathcal{V}[t_{\bullet_j} \to q]} \tilde{q}'_1, p, \tilde{q}'_2 \tag{2}$$

where $s = a^{s_{\mathrm{br}}} :_1 s_1 :_2 \cdots :_n s_n$.

We proceed by a case distinction on the rule θ which led to the introduction of the new transition. In each case, let $t_2 \in \theta(t_1)$ be the result of applying θ at node $t_{\bullet_j} i$. In all cases except when θ removes nodes, q already has an incoming transition, hence we do not need to argue non-redundancy (since $\mathcal{T}$ is non-redundant).

- When $\theta = p' \xrightarrow{\mathrm{rew}_{b \to a}} p$ we derived the new transition from some transtion

  $$q \leftarrow_{i/m} (p', b, R^{\mathrm{new}}_{\mathrm{br}}, R^{\mathrm{new}}_1, \ldots, R^{\mathrm{new}}_n)$$

  and since this transition is sound $t_2 \models_{\mathcal{V}[t_{\bullet_j} \to q]} \tilde{q}'_1, p, \tilde{q}'_2$. We take the run witnessing soundness for t_2 and prepend the application of θ to t_1. This gives us a run witnessing (2) as required.

- When $\theta = p' \xrightarrow{\mathrm{push}_1^k} p$, then when $k > 1$ we derived the new transition from some

$$q \leftarrow_{i/m} (p', a, R_{\mathrm{br}}, R_1, R_2, \ldots, R_n)$$

and $R_1 \xrightarrow[R'_{\mathrm{br}}]{a} R'_1$ and the new transition is of the form

$$q \leftarrow_{j/m} (p, a, R'_{\mathrm{br}}, R'_1, R_2, \ldots, R_{k-1}, R_k \cup R_{\mathrm{br}}, R_{k+1}, \ldots, R_n)$$

Furthermore, we have t_2 has at $t_{\bullet_j} i$ the stack

$$a^{s_k} :_1 a^{s_{\mathrm{br}}} :_1 s_1 :_2 \cdots :_n s_n$$

and we have $s_k \models R_k^{\mathrm{new}} = R_k \cup R_{\mathrm{br}}$ and $s_1 \models R_1^{\mathrm{new}} = R'_1$ and from soundness of $R_1 \xrightarrow[R'_{\mathrm{br}}]{a} R'_1$ we have $a^{s_{\mathrm{br}}} :_1 s_1 \models R_1$. Thus, we can apply soundness of the transition from p' to obtain $t_2 \models_{\mathcal{V}[t_{\bullet_j} \to q]} \tilde{q}'_1, p', \tilde{q}'_2$. We prepend to the run witnessing this property an application of θ to t_1 at node $t_{\bullet_j} i$ to obtain a run witnessing (2) as required.

When $k = 1$ we began with a transition

$$q \leftarrow_{i/m} (p', a, R_{\mathrm{br}}, R_1, R_2 \ldots, R_n)$$

and $R_1 \xrightarrow[R'_{\mathrm{br}}]{a} R'_1$ and the new transition is of the form

$$q \leftarrow_{j/m} (p, a, R'_{\mathrm{br}}, R'_1 \cup R_{\mathrm{br}}, R_2, \ldots, R_n) \ .$$

Furthermore, we have t_2 has at $t_{\bullet_j} i$ the stack

$$a^{s_1} :_1 a^{s_{\mathrm{br}}} :_1 s_1 :_2 \cdots :_n s_n$$

and we have $s_1 \models R_1^{\mathrm{new}} = R'_1 \cup R_{\mathrm{br}}$ and from $s_{\mathrm{br}} \models R_{\mathrm{br}}^{\mathrm{new}} = R'_{\mathrm{br}}$ and soundness of $R_1 \xrightarrow[R'_{\mathrm{br}}]{a} R'_1$ we have $a^{s_{\mathrm{br}}} :_1 s_1 \models R'_1$. Thus, we can apply soundness of the transition from p' using $s_1 \models R_{\mathrm{br}}$ (since $s_1 \models R_1^{\mathrm{new}} = R'_1 \cup R_{\mathrm{br}}$) to obtain $t_2 \models_{\mathcal{V}[t_{\bullet_j} \to q]} \tilde{q}'_1, p', \tilde{q}'_2$. We prepend to the run witnessing this property an application of θ to t_1 at node $t_{\bullet_j} i$ to obtain a run witnessing (2) as required.

- When $\theta = p \xrightarrow{\mathrm{push}_k} p'$ we started with a transition

$$q \leftarrow_{i/m} (p', a, R_{\mathrm{br}}, R_1, \ldots, R_n)$$

and $R_k \xrightarrow[R'_{\mathrm{br}}]{a} (R'_1, \ldots, R'_k)$ and the new transition is of the form

$$q \leftarrow_{j/m} \left(p, a, R_{\mathrm{br}} \cup R_{\mathrm{br}}, R_1 \cup R'_1, \ldots, R_{k-1} \cup R'_{k-1}, R'_k, R_{k+1}, \ldots, R_n\right) \ .$$

Let $s' = a^{s_{\mathrm{br}}} :_1 s_1 :_2 \cdots :_{k-1} s_{k-1}$, we have that t_2 has at node $t_{\bullet_j} i$ the stack

$$a^{s_{\mathrm{br}}} :_1 s_1 :_2 \cdots :_{(k-1)} s_{k-1} :_k s' :_k s_{k+1} :_{(k+1)} \cdots :_n s_n \ .$$

Note, by assumption we have $s_1 \models R_1^{\mathrm{new}} = R_1 \cup R'_1$, $\ldots$, $s_{k-1} \models R_{k-1}^{\mathrm{new}} = R_{k-1} \cup R'_{k-1}$ and $s_{\mathrm{br}} \models R_{\mathrm{br}}^{\mathrm{new}} = R_{\mathrm{br}} \cup R'_{\mathrm{br}}$. Thus from soundness of $R_k \xrightarrow[R'_{\mathrm{br}}]{a} (R'_1, \ldots, R'_k)$ we have $s' \models R_k$. Consequently, from the soundness of the transition from p' we have $t_2 \models_{\mathcal{V}[t_{\bullet_j} \to q]} \tilde{q}'_1, p', \tilde{q}'_2$. We prepend to the run witnessing this property an application of θ to t_1 at node $t_{\bullet_j} i$ to obtain a run witnessing (2) as required.

- When $\theta = p \xrightarrow{\text{pop}_k} p'$ we derived the new transition from

$$q \leftarrow_{i/m} (p', r_k, R_{k+1}, \dots, R_n)$$

and the new transition is of the form

$$q \leftarrow_{j/m} (p, a, \emptyset, \emptyset, \dots, \emptyset, \{r_k\}, R_{k+1}, \dots, R_n)$$

The tree t_2 has labelling $t_{\bullet_j} i$ the stack $s' = s_k :_{(k+1)} \cdots :_n s_n$ and since $s_{k+1} \models R_{k+1}, \dots, s_n \models R_n$ we have from the definition of $\models_{\mathcal{V}}$ and $s_k \models r_k$ that $t_2 \models_{\mathcal{V}[t_{\bullet_j} \to q]} \tilde{q}_1', p', \tilde{q}_2'$. As before, we prepend to the run witnessing this property an application of θ to t_1 at node $t_{\bullet_j} i$ to obtain a run witnessing (2) as required.

- When $\theta = p \xrightarrow{\text{collapse}_k} p'$ we began with a transition

$$q \leftarrow_{i/m} (p', r_k, R_{k+1}, \dots, R_n)$$

and the new transition has the form

$$q \leftarrow_{j/m} (p, a, \{r_k\}, \emptyset, \dots, \emptyset, R_{k+1}, \dots, R_n)$$

The tree t_2 has labelling $t_{\bullet_j} i$ the stack $s' = s_{\text{br}} :_{(k+1)} s_{k+1} :_{(k+2)} \cdots :_n s_n$ and since $s_{k+1} \models R_{k+1}, \dots, s_n \models R_n$ we have from the definition of $\models_{\mathcal{V}}$ and $s_{\text{br}} \models r_k$ that $t_2 \models_{\mathcal{V}[t_{\bullet_j} \to q]} \tilde{q}_1', p', \tilde{q}_2'$. As before, we prepend to the run witnessing this property an application of θ to t_1 at node $t_{\bullet_j} i$ to obtain a run witnessing (2) as required.

- When $\theta = p \xrightarrow{+} (p_1, \dots, p_{m'})$ we had transitions

$$q \leftarrow_{i/m} (q', a, R_{\text{br}}, R_1, \dots, R_n)$$

and

$$q' \leftarrow_{1/m'} \left(p_1, a, R_{\text{br}}^1, R_1^1, \dots, R_n^1\right), \dots, q' \leftarrow_{m'/m'} \left(p_{m'}, a, R_{\text{br}}^{m'}, R_1^{m'}, \dots, R_n^{m'}\right)$$

and the new transition added is of the form

$$q \leftarrow_{i/m} (p, a, R_{\text{br}}^{\text{new}}, R_1^{\text{new}}, \dots, R_n^{\text{new}})$$

where $R_{\text{br}}^{\text{new}} = R_{\text{br}} \cup R_{\text{br}}^1 \cup \dots \cup R_{\text{br}}^{m'}$ and for all k, we have $R_k^{\text{new}} = R_1 \cup R_k^1 \cup \dots \cup R_k^{m'}$. Letting $t_1' =$

$$t +_j (q_1, s_1'), \dots, \left(q_{i-1}, s_{i-1}'\right), (q', s), \left(q_{i+1}, s_{i+1}'\right), \dots, (q_m, s_m')$$

and $\mathcal{V}' = \mathcal{V}\left[t_{\bullet_j} \to q\right]$ we have from $R_{\text{br}}^{\text{new}} = R_{\text{br}} \cup R_{\text{br}}^1 \cup \dots \cup R_{\text{br}}^{m'}$ and $R_1^{\text{new}} = R_1 \cup R_1^1 \cup \dots \cup R_1^{m'}, \dots, R_n^{\text{new}} = R_n \cup R_n^1 \cup \dots \cup R_n^{m'}$, and by soundness of the transition from q' that $t_1' \models_{\mathcal{V}'} \tilde{q}_1', q', \tilde{q}_2'$. Thus, from non-redundancy and repeated applications of the soundness of the transition from p_1 to the soundness from $p_{m'}$ (as in the proof of Lemma C.2 (Sound Acceptance)) we have

$$t_2 = t_1' +_{(j+i)} (p_1, s), \dots, (p_{m'}, s) \models_{\mathcal{V}'\left[t_{\bullet_j} i \to q'\right]} \tilde{q}_1', p_1, \dots, p_{m'}, \tilde{q}_2' .$$

We prepend to the run witnessing this property an application of θ to t_1 at node $t_{\bullet_j} i$ to obtain a run witnessing (2) as required.

The remaining case is for the operations that remove nodes from the tree. For $(p_1, \ldots, p_m) \xrightarrow{\bar{}} p$ we introduced

$$p \leftarrow_{1/m} (p_1, a, \emptyset, \emptyset, \ldots, \emptyset)$$

to

$$p \leftarrow_{m/m} (p_m, a, \emptyset, \emptyset, \ldots, \emptyset) \ .$$

We prove soundness of the first of these rules, with the others being symmetrical. Taking any sequence of transitions

$$p \leftarrow_{1/m} (q_1, r_1), \ldots, p \leftarrow_{m/m} (q_m, r_m)$$

any $t \models_{\mathcal{V}} \tilde{q}_1, p, \tilde{q}_2$ and $s_1, \ldots, s_m$ such that, letting $j = |\tilde{q}_1| + 1$,

$$t' = t +_j (q_1, s_1), \ldots, (q_m, s_m) \models_{\mathcal{V}[t_{\bullet_j} \to p]} \tilde{q}_1, q_1, \ldots, q_m, \tilde{q}_2 \ .$$

We need to show for any stack with top character a that

$$t +_j (p_1, s), (q_2, s_2), \ldots, (q_m, s_m) \models_{\mathcal{V}[t_{\bullet_j} \to p]} \tilde{q}_1, p_1, q_2, \ldots, q_m, \tilde{q}_2 \ .$$

Take the run witnessing the property for t'. This must necessarily pass some tree where $t_{\bullet_j}$ is exposed and contains control state p. Moreover, this is the first such exposure of the node. Since we assume, for all p, there is only one rule $(p'_1, \ldots, p'_2) \xrightarrow{\bar{}} p$ for any $p'_1, \ldots, p'_m$, the node must be exposed by an application of θ.

Thus, we can remove from the run all operations applied to a descendant of $t_{\bullet_j}1$ before its exposure. This run then can be applied to

$$t +_j (p_1, s), (q_2, s_2) \ldots, (q_m, s_m)$$

to witness $t +_j (p_1, s), (q_2, s_2), \ldots, (q_m, s_m) \models_{\mathcal{V}[t_{\bullet_j} \to p]} \tilde{q}_1, p_1, q_2, \ldots, q_m, \tilde{q}_2$.

To prove non-redundancy, we simply take any stacks $s_1, \ldots, s_m$ and apply θ to $t +_j (p_1, s_1), \ldots, (p_m, s_m)$ to obtain t from which the remainder of the run exists by assumption. ☐

Lemma C.6 (Soundness of Saturation). *The automaton $\mathcal{T}$ obtained by saturation from $\mathcal{T}_0$ is such that $\mathcal{L}(\mathcal{T}) \subseteq \mathrm{Pre}^*_{\mathcal{G}}(\mathcal{T}_0)$.*

Proof. By Lemma C.4 (Soundness of $\mathcal{T}_0$) we have that $\mathcal{T}_0$ is sound. Thus, by induction, assume $\mathcal{T}$ is sound. We have $\mathcal{T}' = \mathcal{F}(\mathcal{T})$ and by Lemma C.5 (Soundness of the Saturation Step) we have that $\mathcal{T}'$ is sound.

Thus, the $\mathcal{T}$ that is the fixed point of saturation is sound, and we have from Lemma C.2 (Sound Acceptance) that $\mathcal{L}(\mathcal{T}) \subseteq \mathrm{Pre}^*_{\mathcal{G}}(\mathcal{T}_0)$. ☐

D Lower Bounds on the Reachability Problem

We show that that global backwards reachability problem is n-EXPTIME-hard for an order-n GASTRS. The proof is by reduction from the n-EXPTIME-hardness of determining the winner in an order-n reachability game [8].

Proposition D.1 (Lower Bound). *The global backwards reachability problem for order-n GASTRSs is n-EXPTIME-hard.*

Proof. We reduce from the problem of determining the winner in an order-n pushdown reachability game [8].

We first need to define higher-order stacks and their operations. Essentially, they are just annotated stacks without collapse. That is order-1 stacks are of the form $[a_1 \ldots a_m]_1$ where $a_1 \ldots a_m \in \Sigma^*$. Order-$k$ stacks for $k > 1$ are of the form $[s_1 \ldots s_m]_k$ where $s_1, \ldots, s_m$ are order-$(k-1)$ stacks.

Their operations are

$$\mathrm{HOps}^{\mathrm{a}}_{\mathrm{n}} = \{\mathrm{push}_a \mid a \in \Sigma\} \cup \{\mathrm{push}_k \mid 2 \leq k \leq n\} \cup \{\mathrm{pop}_k \mid 1 \leq k \leq n\} .$$

The push_k and pop_k operations are analogous to annotated stacks. We define $\mathrm{push}_a(s) = a :_1 s$.

Such a game is defined as a tuple $(\mathbb{P}, \Sigma, \mathcal{R}, \mathcal{F})$ where $\mathbb{P} = \mathbb{P}_1 \cup \mathbb{P}_2$ is a finite set of control states partitioned into those belonging to player 1 and player 2 respectively, Σ is a finite set of stack characters, $\mathcal{R} \subseteq \mathbb{P} \times \Sigma \times \mathrm{HOps}^{\Sigma}_{\mathrm{n}} \times \mathbb{P}$ is a finite set of transition rules, and $\mathcal{F} \subseteq \mathbb{P}$ is a set of target control states.

Without loss of generality, we assume that for all $p \in \mathbb{P}_2$ and $a \in \Sigma$ there exactly two rules in $\mathcal{R}$ of the form (p, a, σ, p') for some σ and p'.

A configuration is a tuple (p, s) of a control state and higher-order stack. A winning play of a game from an initial configuration (p_0, s_0) for player 1 is a tree labelled by configurations such that

- all leaf nodes are labelled by configurations (p, s) with $p \in \mathcal{F}$.
- if an internal node is labelled (p, s) with $p \in \mathbb{P}_1$ then the node has one child labelled by (p', s') such that for some $(p, a, \sigma, p') \in \mathcal{R}$ we have $s = a :_1 s''$ for some s'' and $s' = \sigma(s)$.
- if an internal node is labelled (p, s) with $p \in \mathbb{P}_2$ then when $s = a :_1 s'$ for some s' and we have the rules (p, a, σ_1, p_1), and (p, a, σ_2, p_2), then the node has two children labelled by (p_1, s_1) and (p_2, s_2) with $s_1 = \sigma_1(s)$ and $s_1 = \sigma_1(s)$.

Note, we assume that the players can always apply all available rules for a given p and a in the game (unless a control in $\mathcal{F}$ is reached). This is standard and can be done with the use of a "bottom-of-stack" marker at each order.

Determining if player 1 wins the game is known to be n-EXPTIME hard [8]. This amounts to asking whether a winning game tree can be constructed from the initial configuration (p_0, s_0).

That the winning game trees are regular can be easily seen: we simply assert that all leaf nodes are labelled by some $p \in \mathcal{F}$.

We build a GASTRS that constructs play trees. We simulate a move in the game via several steps in the GASTRS, hence its control states will contain several copies of the control states of the game. Suppose we have a rule (p, a, σ, p') where $p \in \mathbb{P}_1$. The first step in the simulation will be to check that the top character is a, for which we will use $p \xrightarrow{\mathrm{rew}_{a \to a}} (p, 1)$ where $(p, 1)$ is a new control state. The next step will create a new node in the play tree using $(p, 1) \xrightarrow{+} ((p', 2))$ which uses the intermediate control state $(p', 2)$. The final step is to apply the stack operation and move to p'. When $\sigma = \mathrm{push}_k$ or $\sigma = \mathrm{pop}_k$ we can use $(p', 2) \xrightarrow{\sigma} p'$. When $\sigma = \mathrm{push}_b$ we use another intermediate control state and $(p', 2) \xrightarrow{\mathrm{push}^1_1} (p', 3)$ and $(p', 3) \xrightarrow{\mathrm{rew}_{a \to b}} p'$.

When $p \in \mathbb{P}_2$ with the rules (p, a, σ_1, p_1) and (p, a, σ_2, p_2) we use $p \xrightarrow{\mathrm{rew}_{a \to a}} (p, 1)$,

$$(p, 1) \xrightarrow{+} ((p_1, 2), (p_2, 2)) ,$$

and similar rules to the previous case to apply σ and move to p_1 or p_2.

Let the above GASTRS be $\mathcal{G}$. From the initial single-node tree t_0 whose node is labelled (p_0, s_0) it is clear that a tree whose leaf nodes are only labelled by control states in $\mathcal{F}$ can be reached iff there is a winning play of player 1 in the reachability game. We can easily build a tree automaton $\mathcal{T}_0$ that accepts only these target trees. Since checking membership $t_0 \in \mathrm{Pre}^*_{\mathcal{G}}(\mathcal{T}_0)$ is linear in the size of tree automaton representing $\mathrm{Pre}^*_{\mathcal{G}}(\mathcal{T}_0)$ we obtain our lower bound as required. $\square$