Report

Eashan Gupta

July 2018

Introduction

The aim of the internship was to define a parity game on higher order recursion scheme and find a reduction of the same to a safety game using the previous results of reductions on games played on collapsible pushdown automata. Also, the other part was to find a lower bound to the number of counters required in the reduction from parity to safety game on CPDS for various cases.

Parity Game on a higher order recursion scheme

We can define a parity game on a recursion scheme G, given by $\langle \sum, N, R, S \rangle$, by $\langle \sum, N, R, S, T, Rk, n \rangle$ where the function $T : N \to \{\text{A, E}\}$ gives the owner of each non-terminal and $Rk : R \to [n]$ gives the ranks of the rewrite rules. The game is defined on the process of generating the value tree for the recursion scheme. Each rewrite rule has a rank given by Rk and the order of application of these rules, to generate the value tree, gives the winner of the game.

The game is played on the computation tree which is used to generate the value tree. At each point in the computation tree, we assign the tree generated till now an owner. The tree generated till now is assigned the same owner as that of its leftmost non-terminal. He can choose any valid applicable rewrite rule from R to apply on the tree and take the game further. The owner is so decided as by observing the reduction from CPDS to HORS, the leftmost non-terminal should be given priority.

Winning Condition

The winning condition is the same as that for parity games played on pushdowns ie the lowest rank occurring infinite number of times should be odd for A to win. Else, E wins.

Safety Game on a higher order recursion scheme

The safety game on a HORS can be defined in the same way by defining some rewrite rules as unsafe and avoiding their use.

Idea to reduction

My idea was to generate a reduction to convert a given higher order recursion scheme (G) to another HORS (G') which would successfully represent the generation process of the value tree of the given

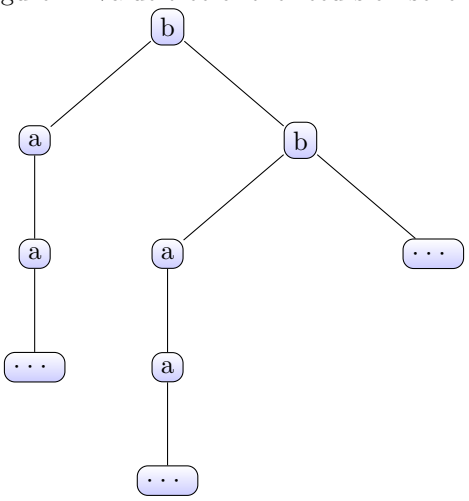
recursion scheme. The generation process of the value tree for any recursion scheme can be surmised in the computation tree. For the following game, we only need to store the ranks of the edges(ie the ranks of the rewrite rule used there) and the owner of various vertices to determine a winning strategy.

So if we are able to generate a new recursion scheme (G') to represent this computation tree T of G (the value tree of G'), we might be able to run simpler model checkers directly on G'.

Example

$$S \to Fc$$
$$Fx \to b(Ha)(Fx)$$
$$Hx \to x(Hx)$$

Figure 1: Value tree of the recursion scheme



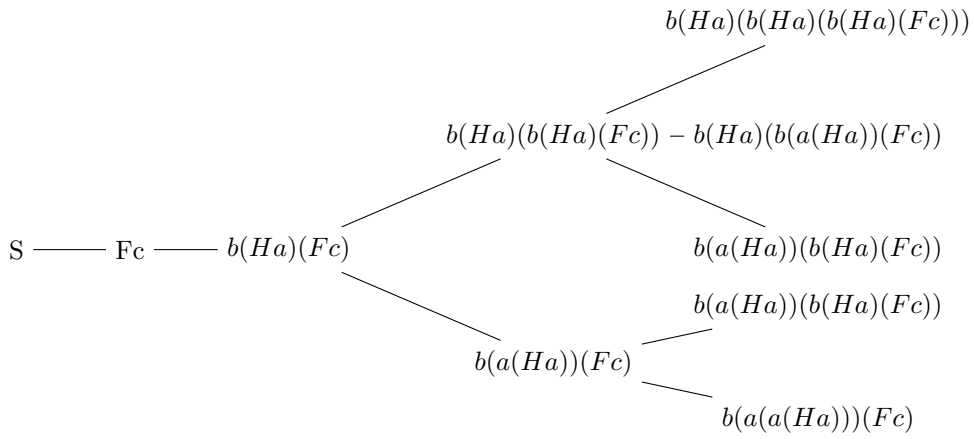
Problem with this approach

The problem with this idea is that the tree grows exponentially as we go on and it would require infinite non-terminals in the worst case to store the tree. Also, at each point to only keep track of how many outgoing edges are there at any vertex, we need to store the structure of the value tree until then and thus the total storage needed blows up to n-fold exponential.

Observation

The aim was to use the already proved reduction from parity to safety game on CPDS and the reduction from HORS to CPDS and vice versa as a basis to get reduction from parity to safety on HORS. But I believe that this might not be easy for the currently defined games on higher order recursion scheme. According to the equivalence between HORS and CPDS, it seems that the parity game on a CPDS corresponds to a game played on the value tree of the recursion scheme. This

Figure 2: Computation tree of the recursion scheme to generate the value tree



can be observed from the way the reduction is defined and superposing a game on this. There is no attractive way of defining a parity game on a recursion scheme based only on its value tree. So, though this game defined on the computational tree is attractive, it might have to be viewed or solved differently.

Using Krivine Machines

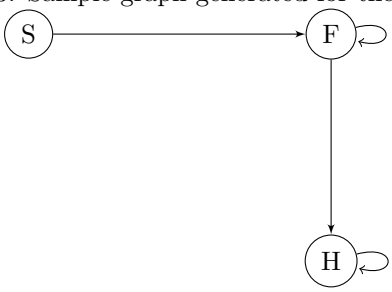
Another way to observe the generation process of the value tree for a recursion scheme maybe using the Krivine machines. But this tool too uses the value tree as the basis. It only defines things on the value tree and I was not able to relate the parity game as defined above to it.

Use a Game on a graph instead of trees

Another way to summarize the generation process is to use a digraph, $\langle V, E \rangle$ generated using the rewrite rules. Each non-terminal is represented by a vertex in V. For any rewrite rule, take the head non-terminal on the left side and draw edges from the vertex representing it to the vertices represented by all non-terminals present on the right side of the rewrite rule. By passing tokens, we can play a game on this graph to represent a parity game as defined above. The tokens are passed from one vertex to all the descendents and the player who will play that turn is determined by a special token.

The problem with this idea is that hierarchies between tokens will have to be maintained ie moving one token will have to increase the number of tokens at some very different vertex if we view it using the value tree. As using a rewrite rule in a sequence may increase the occurrence of some other non-terminal, we will have to increase the number of tokens for it too. This will have to be tracked using a tree of a very large size (something similar to the computation tree or the value tree).

Figure 3: Sample graph generated for the example



Lower bounds for the reduction

In the reduction from parity to safety, we first use the Walukiewicz reduction to convert it into a finite state parity game of exponential size. Our previous lower bound is n-fold exponential ie it cannot go worse than that. Thus our aim is to limit the complexity on the reduced graph (of exponential size).

So I started with a graph and have been trying to use graph properties like observing cycles but have not been able to come up with something concrete. I believe we can use some graph properties directly to simplify the bound. The problem is that there is no bound on the ranks assigned to configurations. For the same graph, just changing the ranks of vertices (configurations) will change the winning strategy.