GAME SEMANTICS OF HIGHER-ORDER RECURSION SCHEMES ESTABLISHES THE DECIDABILITY OF MSO MODEL-CHECKING

Charles Grellois[*]

Abstract

This article presents two different ways of model-checking higher-order recursion schemes, both relying on *game semantics*. A given recursion scheme is translated to another, which is its *computational extent*, in the sense that β-reduction paths - called *traversals* - in the new generated tree are isomorphic to branches of the former tree. Then, the two approaches differ on their way of simulating traversals: one uses APTs to produce verification parity games, whereas the other uses n-CPDAs and builds verification parity games on their transition graphs. Both approaches prove that the modal μ-calculus model-checking of an order-n recursion scheme is n-EXPTIME complete.

Introduction

With the development of Computer Science, the need for reliable computing systems is growing; but the complexity of programs and systems keeps increasing. Verifying programs is crucial; and several approaches were found so far. *Model-checking*, introduced in the early 80s, emphasizes on the verification of a given *model* of a program, instead of trying to verify the whole program which may be millions lines long. The problem is then, given a model of a program which sketches its behaviour, and a logical formula expressing a property the program should have (like, not entering in a bad state), to check whether the formula holds on the model.
Modal μ-calculus was introduced in [8] (1982); it is a temporal logic convenient for the verification over branching structures, like trees, and adapted to the characterization of recursive properties thanks to fixpoint operators.

[*]Student of the ENS Cachan (France) - `charles@grellois.fr`.

Higher-order recursion schemes were introduced in the 70s, and provide a convenient way to model functional programs generating trees, with recursion but without side effects. Interest for these schemes has been raising in the recent years, and Ong gave in [9] a way to transform such schemes so that their dynamics, from *syntactic*, become *semantic*, being given then by β-reductions. Model-checking of a modal μ-calculus may then be performed on the transformed scheme, by following call-by-name β-reduction paths called *traversals*, which compute the branches of the tree generated by the original scheme. Ong introduced two different ways of simulating these traversals for verificational purpose, in [9] and [5], the first one using alternating parity automata, and the second one using higher-order collapsible pushdown automata. These two approaches are introduced in this article, with some additionnal information on modal μ-calculus.
This work was made during an internship in Oxford; my contribution is only about explaining things and adding extended or new examples. No new result is presented in this article.

Acknowledgements. First of all I would like to thank Luke Ong for his advices and comments, and his group for support and interesting discussions. I would also like to thank Paul-André Melliès for introducing me to the field and helping in my first reading of [9], and William Blum for his very useful software *HOG*[1], which allows to automate most of the processes described in this paper and to export them to LaTeX.

Contents

[1]http://william.famille-blum.org/research/tools.html

1 Modeling programs: Recursion Schemes

1.1 Higher-Order Recursion Schemes

The first step of the verification of a given program is to define a proper model of it. Higher-order recursion schemes provide a way to model programs generating trees, with the expressivity of a simply-typed λ-calculus with recursion and uninterpreted constants[2], and we shall see that they also are a proper modelling structure for verification. Before defining these recursion schemes, we first need to introduce their type system.

Types for the recursion schemes. In the following we just consider a ground type o, and applicative types based on it. Thus every type will have a decomposition $A_1 \to \cdots \to A_n \to o$; n is the *arity* of the type, and its *order* is defined by:

- $order(o) = 0$
- $order(A_1 \to \cdots \to A_n \to o) = 1 + \max(order(A_1), \cdots, order(A_n))$

Higher-Order recursion schemes. A recursion scheme is some kind of “grammar with parameters” which generates a tree, the formal definition being:

Definition 1. A recursion scheme is a 4-tuple $G =< \Sigma, \mathcal{N}, \mathcal{R}, S >$, with Σ a ranked alphabet of *terminals*, $\mathcal{N}$ a finite set of *non-terminals*[3], $S \in \mathcal{N}$ a *start symbol* of type o, and $\mathcal{R}$ a finite set of *rewrite rules*, one for each non-terminal $F : A_1 \to \cdots \to A_n \to o$, of the shape $F\xi_1 \cdots \xi_n \to e$ where the $\xi_i : A_i$ and $e : o$.

[2]Of order at most 1; the notion will be defined soon. These constants will be the nodes of the generated tree.

[3]In the following, we use upper-case letters for the non-terminals, and lower-case ones for the terminals.

The *order* of a recursion scheme is then defined to be the maximal order of its non-terminals[4]. A recursion scheme G defines a term and thus a tree, denoted $[\![G]\!]$, obtained by unfolding the rewrite rules of G *ad infinitum*. The generated tree may be infinite[5]. Intuitively, the order of a recursion scheme describes the complexity of the generated tree, order-0 recursion schemes generating regular trees.

1.2 Giving a computational structure to the recursion schemes: the RHS transform

Such a recursion scheme does not have a convenient structure for our purpose; here we describe the RHS transform which gives an equivalent recursion scheme which basically is the *computational extent* of the former: the new recursion scheme will be some kind of λ-calculus program computing the branches of the tree.

Definition 2. The RHS transform of a recursion scheme G is a four-step rewriting of the rules of G, leading to a new recursion scheme $\bar{G}$:

1. η-expand every subterm which occurs in operand position, even if it has ground type.
2. Insert apply-symbols @ such that every application $De_1 \cdots e_n$ becomes $@De_1 \cdots e_n$
3. Curry the rewrite rules: $F\xi_1 \cdots \xi_n \to e$ becomes $F \to \lambda\xi_1 \cdots \xi_n.e$ (even when $n = 0$ where this amounts to add a $\lambda.$ on the right side).
4. α-rename the bound variables.

The new recursion scheme then generates the *computation* tree denoted $\lambda(G)$, which is the η-long Böhm tree of $[\![G]\!]$. This tree is enriched with a notion of *direction*: to each child of a given node is assigned a direction, the leftmost child being given the direction 0 if the node is labelled by an @ and 1 else; the direction is increased by one for the child immediatly right of the leftmost one, and so one.

[4]Intuitively, the order of a type describes the maximal depth of imbricated beta-reductions it needs to compute. For example, to compute an order-2 term, one will have to β-reduce once at least one of its arguments; the second β-reduction being the one of the term itself, that is performed by the rewriting system.

[5]Equivalently, one can consider the graph generated by a recursion scheme, obtained by considering first a forest whose trees correspond to the applicative terms on the right of the rules, and where links to a given non-terminal are added as edges to the root of the corresponding tree. The unfolding of this (finite) graph is then the (potentialy infinite) generated tree.

This way, the code of an application is stored in direction 0 and the code of the i^{th} argument in direction i. We additionally define $\mathrm{Dir}(f)$ to be the set of directions of a given symbol f.

Local structure of $\lambda(G)$. The RHS transform gives a very convenient structure to $\lambda(G)$: locally, the tree is of one of the following forms:

- An applicative node of arity n has $n + 1$ children:

$$\begin{tikzcd}
 & @ \arrow[dl, no head] \arrow[d, no head] \arrow[drr, no head] & & \\
\lambda\xi_1\cdots\xi_n & \lambda\bar{\eta}_1 & \cdots & \lambda\bar{\eta}_n
\end{tikzcd}$$

  where the leftmost node, in direction 0, will be the root of the application's code, where ξ_i describes the i^{th} argument of the application, and where the child in direction $i \geq 1$ corresponds to the code of the i^{th} argument of the application (so, in a *call-by-name* strategy, encountering ξ_i in the applicative code leads to computing the branch in direction i). $\bar{\eta}_i$ then describes the list of (potential: the node may just be labelled λ if ξ_i has order 0) arguments of ξ_i, which may have themselves to be accessed during the β-reduction of ξ_i if ξ_i has order at least 2.

- A node labelled by a variable ϕ of arity n has n children, in directions 1 to n:

$$\begin{tikzcd}
 & \phi \arrow[dl, no head] \arrow[dr, no head] & \\
\lambda\xi_1 & \cdots & \lambda\xi_n
\end{tikzcd}$$

  where the i^{th} direction contains the code of the i^{th} argument of ϕ.

- A Σ-labelled node of arity n has n children:

$$\begin{tikzcd}
 & f \arrow[dl, no head] \arrow[dr, no head] & \\
\lambda & \cdots & \lambda
\end{tikzcd}$$

  If $n = 0$, the node is terminal.

Example 1 (From G to $\lambda(G)$)**.** We consider the following simple order-2 recursion scheme G:

$$S \to H\ a$$
$$H\ z \to F\ (g\ z)$$
$$F\ \phi \to \phi\ (\phi\ (F\ h))$$

Applying the RHS transform, one obtains the following scheme:

$$S \to \lambda.@\ H\ (\lambda.a)$$
$$H \to \lambda z.@F(\lambda y.g(\lambda.z)(\lambda.y))$$
$$F \to \lambda\phi.\phi(\lambda.\phi(\lambda.@\ F\ (\lambda x.h(\lambda.x))))$$

One should remark that the new recursion scheme is order-0 thanks to the curryfication, has an explicit marking of the applications with the use of the @ symbol - but paths in $\lambda(G)$ have no relation with paths in $[\![G]\!]$. The notion of traversal will give such a relation.
Figure 1 gives the two trees corresponding to G and its transform.

1.3 Preservation by RHS transform: the Path-Traversal correspondence

Traversals: informal ideas. The idea of the RHS transform is to turn a recursion scheme into some λ-calculus term-generating scheme, such that the computation by β-reduction of the generated term computes the original tree, when one erases the non-Σ symbols.
Thus, a path in the tree $[\![G]\!]$ will correspond to a computational sequence in the tree $\lambda(G)$; we call this sequence a *traversal*. The idea of a traversal is that it starts from the root of $\lambda(G)$, then goes down. When a @ symbol is reached, the 0-child is visited first: remember that it is the code of the application. Then when the value of the i^{th} argument is required, the traversal jumps to the i^{th} child of the application node, and computes the value of this argument, then jumps back to what follows in the applicative code[6]. When a Σ-symbol of arity non-zero is reached, the environment chooses a direction to compute: thus there will be one (maximal) traversal per branch of $[\![G]\!]$.

Traversals: formal definition. We need some more definitions first. The first one, defining the *enabling relation*, and *justified sequences*, will give us control over the β-jumps in the tree. The second one, defining the *view* of justified sequence, has a game semantics reason of being, which will be explained in the next paragraph. *Traversals* will then be defined as some justified sequences, constrained by their view.

[6]Thus the traversals follow a call-by-name reduction strategy: given that we do not modify the tree but only walk on it, it is the only solution.

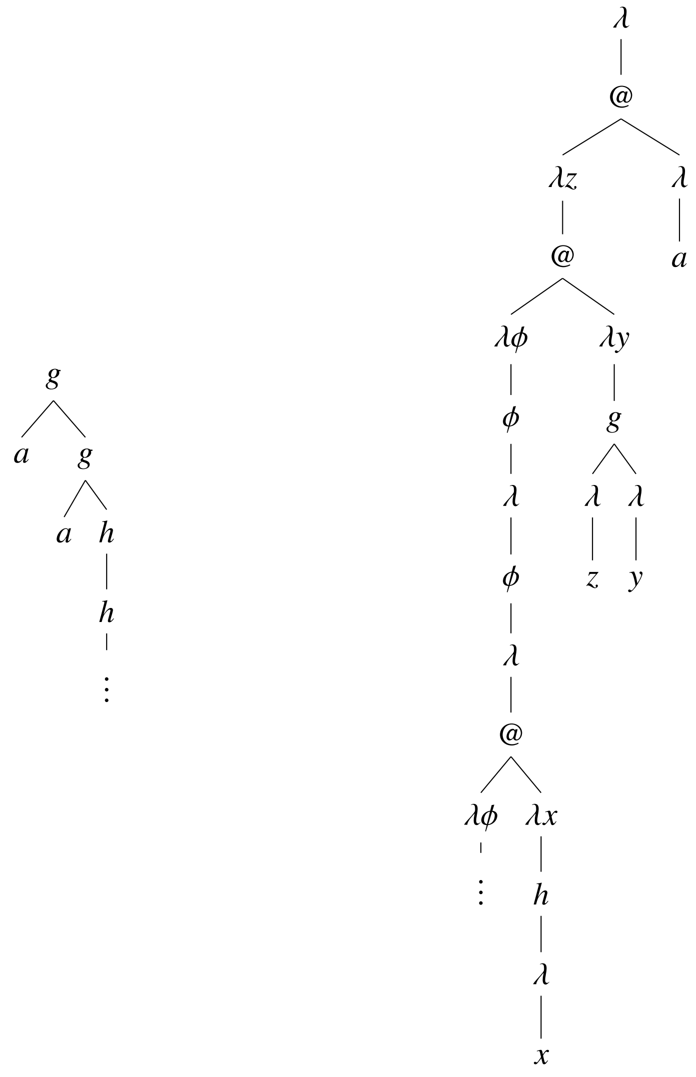


Figure 1: $[\![G]\!]$ and $[\![\lambda(G)]\!]$

Definition 3 (Enabling relation, justified sequences)**.** The *enabling relation* is defined on the nodes of the computation tree:

- Every λ node but the root is i-enabled by its parent node, where i is its direction
- Every variable node ξ_i is i-enabled by its binder $\lambda\bar{\xi}$ (where ξ_i is the i^{th} element of the list $\bar{\xi}$)

A *justified sequence* is then be a sequence of nodes alternating λ and non-λ nodes, such that if a node i-enabled by another appears, then its enabler appears before in the sequence.

We represent the enabling relation by justification links, as in the following example over $\lambda(G)$:

$$\begin{tikzcd}[column sep=small]
\lambda & @ & \lambda z \arrow[l, bend right, "0"'] & @ & \lambda\phi \arrow[l, bend right, "0"'] & \phi \arrow[l, bend right, "1"'] & \lambda y \arrow[lll, bend right, "1"'] & g & \lambda \arrow[ll, bend right, "1"'] & y \arrow[lll, bend right, "1"'] & \lambda
\end{tikzcd}$$

Definition 4 (View of a justified sequence)**.** The view $\ulcorner t \urcorner$ of a justified sequence t is defined by induction as follows:

- $\ulcorner \lambda \urcorner = \lambda$
- $\begin{tikzcd}[column sep=tiny] \ulcorner t & \lambda\bar{\xi} & \ldots & y \arrow[ll, bend right, "i"'] & \urcorner = \ulcorner t \urcorner & \lambda\bar{\xi} & \ldots & y \arrow[ll, bend right, "i"'] \end{tikzcd}$
- $\ulcorner t\lambda\bar{\xi}n \urcorner = \ulcorner t\lambda\bar{\xi} \urcorner n$

where in the third rule a link is added to the view if there was one from n to a previous element appearing in the view.

Definition 5 (Traversal)**.** A traversal is a justified sequence generated incrementally by the five following rules:

1. The justified sequence made of the root node of $\lambda(G)$ is a traversal.
2. If t @ is a traversal, then $\begin{tikzcd}[column sep=tiny] t & @ & \lambda\bar{\phi} \arrow[l, bend right, "0"'] \end{tikzcd}$ is one too.
3. If $t\ f$ (with $f \in \Sigma$) is a traversal then, for every $1 \leq f \leq arity(f)$, $\begin{tikzcd}[column sep=tiny] t & f & \lambda \arrow[l, bend right, "i"'] \end{tikzcd}$ is a traversal too.
4. If $\begin{tikzcd}[column sep=tiny] t\ n & \lambda\bar{\xi} & \cdots & \xi_i \arrow[ll, bend right, "i"'] \end{tikzcd}$ is a traversal, $\begin{tikzcd}[column sep=tiny] t & n & \lambda\bar{\xi} & \cdots & \xi_i \arrow[ll, bend right, "i"'] & \lambda\bar{\eta} \arrow[llll, bend right, "i"'] \end{tikzcd}$ is one too[7].
5. If $t\lambda\bar{\xi}$ is a traversal, and $\ulcorner t\lambda\bar{\xi}n \urcorner$ is a path in $\lambda(G)$, then $t\lambda\bar{\xi}n$ is a traversal too.

[7]Here we see the interest of the enabling relation: together with this rule, it ensures that, in order to compute an i^{th} argument, the traversal jumps to the code of it and nowhere else.

Most rules meet the informal explanation that was given at the beginning of this subsection; the only thing which needs more explanations is how a traversal jumps back to the right node after computing an argument. That is the purpose of the fifth rule, and of the notion of *view*: the view hides the internal computations that are made, typically here the computation of an argument's value. Then, if n is the son of the node ϕ where the traversal jumped from, the view of the traversal will end with $\cdots \phi n$, and by construction the whole view will be the path from the root node to n in $\lambda(G)$.

Example 2 (Traversals of $\lambda(G)$)**.** We consider again the scheme of the previous example, and compute some traversals over it.
First, we have to start with the root node; and then, we walk down on the tree, always taking the direction 0 after visiting an applicative node. Thus we obtain the following traversal:

$$\begin{tikzcd}\lambda & @ & \lambda z \arrow[l, bend right, "0"'] & @ & \lambda\phi \arrow[l, bend right, "0"'] & \phi \arrow[l, bend right, "1"']\end{tikzcd}$$

, which is just a path in $\lambda(G)$. Now that we arrived on ϕ, we need to compute its value, that is the first (and only) argument of the application that enables its binder (the second @ we have met so far). So, the traversal grows jumping to the son in direction 1 of this second @, and continues going down in this subtree, leading to

$$\begin{tikzcd}\lambda & @ & \lambda z \arrow[l, bend right, "0"'] & @ & \lambda\phi \arrow[l, bend right, "0"'] & \phi \arrow[l, bend right, "1"'] & \lambda y \arrow[lll, bend right, "1"'] & g\end{tikzcd}$$

.
g is a Σ-symbol: here the two directions may be taken to build a traversal[8].

- If we choose to visit direction 1, the traversal trivially grows to:

$$\begin{tikzcd}\lambda & @ & \lambda z \arrow[l, bend right, "0"'] & @ & \lambda\phi \arrow[l, bend right, "0"'] & \phi \arrow[l, bend right, "1"'] & \lambda y \arrow[lll, bend right, "1"'] & g & \lambda \arrow[l, bend right, "1"'] & z \arrow[lllllll, bend right, "1"']\end{tikzcd}$$

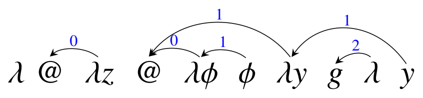
  z is a variable node, corresponding to the first and only argument of the first application we have met, so the traversal has to jump to the son in direction 1 of the first @, growing then to:

$$\begin{tikzcd}\lambda & @ & \lambda z \arrow[l, bend right, "0"'] & @ & \lambda\phi \arrow[l, bend right, "0"'] & \phi \arrow[l, bend right, "1"'] & \lambda y \arrow[lll, bend right, "1"'] & g & \lambda \arrow[l, bend right, "1"'] & z \arrow[lllllll, bend right, "1"'] & \lambda a \arrow[lllllllll, bend right, "1"']\end{tikzcd}$$

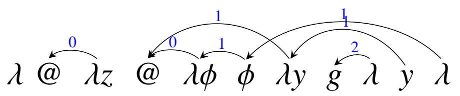
[8]It corresponds to computing the branches of $[\![G]\!]$, choosing one of the two possible directions after visiting g.

which is a maximal traversal[9].

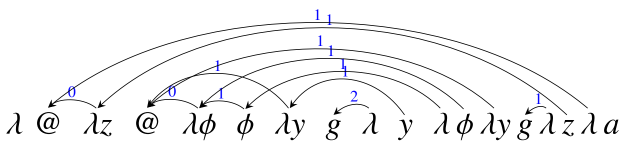
- If we choose to visit direction 2, the traversal trivially grows to:



Then the traversal has to grow to:



where this λ is the son of the ϕ node which the traversal jumped from, because the only rule applying here is the 5th one, and that this justified sequence's view is a path in $\lambda(G)$: thus it is a traversal. Going down again adds a new occurence of ϕ to the traversal, and it jumps again to the same subtree; if after visiting g the first direction is chosen, the maximal traversal jumps again to the subtree with an occurence of a a[10], leading to:



else the traversal jumps back to a new application node, ...[11].

The reader may refer to [4, Appendix A] for an order 3 example, where a β-jump is sometimes performed inside a β-jump.

[9]One may remark that this traversal visited g, then a, which is one of the maximal branches of $[\![G]\!]$.

[10]So the maximal traversal will be finite and correspond to the path gga in $[\![G]\!]$

[11]The maximal traversal in this direction is infinite and corresponds to the only infinite path in $[\![G]\!]$: $gghh\cdots$

Traversals and games. The tree $\lambda(G)$ was built such that λ and non-λ nodes alternate[12]; this gives $\lambda(G)$ a structure of *arena*, with λ nodes belonging to the *Opponent* and non-λ nodes belonging to the *Proponent*. The traversals may now be constructed as plays[13] over this arena, where the only non-deterministic operation happens on the Σ nodes: on this node, the proponent (that is, the program) asks the opponent (the environment) which is the next node to visit, and the opponent answers by picking one of its λ-nodes, that is by choosing the direction in $[\![G]\!]$ to compute. The succession of such choices will build a branch of $[\![G]\!]$ if one erases the computation symbols added to build $\lambda(G)$: this is the path-traversal correspondence.

The Path-Traversal correspondence. Using (innocent[14]) game semantics, the following holds:

Theorem 1. *There is a 1-to-1 correspondence between maximal paths in the tree $[\![G]\!]$ and maximal traversals in the tree $\lambda(G)$; and the Σ-symbols on the traversal are exactly the Σ-symbols on the corresponding path of $[\![G]\!]$.*

This justifies the former affirmation that traversals *compute* $[\![G]\!]$: each maximal traversal computes a branch of the original tree if we remove every non-Σ symbol added for the computation purposes, as we have seen on the example while computing the three maximal traversals of $\lambda(G)$.

Remark 1.

- In the example we computed traversals on an order-2 recursion scheme. The order of a recursion scheme is intuitively the maximal *depth* of jumps in the tree plus one; so in an order-3 recursion scheme, traversals sometimes jump a first time to simulate the computation of an argument, and a second one to perform a β-reduction needed by the computation of this argument; it may have to perform several such depth-2 jumps sequentially. An example of an order-3 recursion scheme, of it RHS-transform, its computation tree, its traversals, ... is given in [4, Appendix A]. If you refer to this, you may remark that at order 3, the infinite direction is not necessary the leftmost one, even if it may seem in the examples we have seen so far.

[12]This is the reason why even ground-type subterms were η-expanded, and even the rules with no arguments were curryed during the RHS transform.

[13]The definition of a traversal gives a *strategy* whose plays are the traversals. Note that this strategy only uses its *view* to determinate what move to play next: such a strategy is said to be *innocent*; and innocent strategies represent programs without side effects.

[14]That is, game semantics where the strategies are only based on the view: when two moves have a similar view, the next move will be the same in both case. This basically means that there are no side effects.

- Now that we have this computational intuition of the RHS transform of a recursion scheme, one should understand that the α-renaming performed during the transform ensures that, in a *local* context, which is respected by the traversal's jumps, a given variable symbol represents one unique, well-defined value, even if in the whole tree infinitely many occurences of the symbol corresponding to different values in a computation may appear.

2 Modal μ-calculus, verification games, and traversals

2.1 Modal μ-calculus

An informal definition. Modal μ-calculus is some kind of temporal logic; it essentially is an enrichment of usual Boolean formulae with two fixpoint operators μ (smallest fixpoint operator) and ν (greatest fixpoint operator), and two modalities $\Box$ (which requires that every immediate[15] successor of the current temporal position satisfies the subformula it applies to) and $\diamond$ (which just requires one of the immediate successors to satisfy the subformula it applies to)[16].
For our purpose of performing model-checking over the value tree of a recursion scheme, the branching structure will be the tree, whose edges are considered directed "from the root to the leaves": therefore, $\diamond\phi$ is true iff ϕ holds at some son of the current node, and $\Box\phi$ is true iff ϕ holds at every son of the current node.
The two fixpoints μ and ν can be interpreted as two *recursion operators*; μ allowing only *finite* recursion, and ν allowing infinite recursion.[17]

Example 3 (Modal μ-calculus formulae over a tree)**.** Here we present several modal μ-calculus formulae, of increasing complexity:

- $\mu Z.(\Box Z)$ is true iff the tree is finite: the checking of this formula starts at the root of the tree, and the formula iterates itself (with a μ operaror, prohibiting infinite recursion) on the children of the root, $\cdots$

[15]Contrary to what happens with "Always" and "Eventually" in CTL, these modalities only affect the immediate successors. But, as we shall see soon, the combination of these modalities with fixpoints, which provide *recursion*, allows to encode these two CTL modalities in the modal μ-calculus.

[16]These ideas should be enough for what follows; for "real" definitions, one may refer to [2], or to [11] which provides a good intuition - but uses a slightly different model of APT than ours.

[17]The idea is that the smallest fixpoint of a function f over a lattice can be obtained by iterating f on $\perp$. The process necessarily ends; up to a little refinment, within a finite number of iterations. See [2], sections 3.1 and 3.2, for more explanations - in the following, these informal explanations should suffice.

- $\mu Z.(a \vee \diamond Z)$ is true iff some branch of the tree has a a label on it: either there is an "a" at the current node, either $\diamond Z$ holds, that is $\mu Z.(a \vee \diamond Z)$ holds at one children (at least) of the current node; the process then iterates on this node - note that the use of μ implies that it stops at some point, where necessarily a labels the node. If a ν operator had been used, the modelled property would have been "there is an a on a branch of the tree or an infinite branch".
- What if we want to characterize the existence of the infinite branch with an infinity of h we had in the value tree of Example 1 ? We have to model the property that there is a path which, after a finite number of nodes, reaches a node where starts one branch, whose labelling word is $hhh\cdots$ This branch property can be expressed by $\psi = \nu Y.(h \wedge \diamond Y)$, which means that h labels the current node, and that ψ itself holds at one child of the current node: by infinite iteration, this formula holds at a node iff it is the start of a branch labelled $hhh\cdots$ To reach this branch, we use the formula $\phi = \mu Z.(\psi \vee \diamond X)$, which expresses that, within a finite number of steps, some path leads to a node where ψ holds: thus we designed in ϕ the property we wanted to.

Remark 2 (CTL is embedded in the modal μ-calculus.)**.** To translate a CTL formula into a modal μ-calculus one, it should be clear that the difficulty is just to translate the two operators $\forall[\phi\mathbf{U}\psi]$ and $\exists[\phi\mathbf{U}\psi]$. Remember that the first one means that for every infinite path starting from the current node, ϕ holds until, within a finite number of steps, one reaches a node where ψ holds. The second one means that there exists a finite path starting from the current node in which ϕ holds until some point where ψ does.
In the first case, we just want to iterate finitely the formula $\psi \vee \Box\phi$, leading to the μ-calculus formula $\mu Z.(\psi \vee (\Box\phi \wedge Z))$: intuitively, if this formula holds at some node, then on every explored direction, either ψ holds, and the exploration stops, or ϕ holds at every son of the node, and the iteration starts the exploration at every son[18]. The iteration being finite by the use of μ, we have the expected semantics.
In the second case, the corresponding formula is $\mu Z.(\psi \vee (\diamond\phi \wedge Z))$ for the same reasons.

2.2 Modal μ-calculus model-checking on trees using alternating parity automata

Alternating parity automata. Alternating parity automata are a generalization of non-deterministic tree automata. We shall see that their behaviour is perfect for checking whether a given modal μ-calculus formula holds at the root of a given

[18]See [2], section 4.1, for a formal translation.

tree.

Definition 6 (Alternating Parity Automaton)**.** An alternating parity automaton (APT) is a tuple $< \Sigma, Q, \delta, q_0, \Omega >$ where:

- Σ is the (ranked) alphabet, whose elements label the nodes of the input trees
- Q is the finite set of states
- δ is the transition function, which maps to each $(q, f) \in Q \times \Sigma$ a positive Boolean formula whose atoms are couples from $\mathrm{Dir}(f) \times Q$
- q_0 is the initial state
- $\Omega : Q \to \mathbb{N}$ is the priority function

The generalization of non-determinism comes from the fact that instead of having two possible transitions $\delta(q, f) = (q_1, \cdots, q_n)$ and $\delta(q, f) = (q'_1, \cdots, q'_n)$ from some configuration (q, f), in an APT one will have:

$$\delta(q, f) = ((1, q_1) \wedge \cdots \wedge (n, q_n)) \vee ((1, q'_1) \wedge \cdots \wedge (n, q'_n)).$$

Definition 7 (Run-tree of an APT)**.** A run-tree of an APT $\mathcal{A}$ over a Σ-labelled ranked tree t is an unraked tree r such that

- $\epsilon \in \mathrm{Dom}(r)$ and $r(\epsilon) = (\epsilon, q_0)$
- For every $\beta \in \mathrm{Dom}(r)$ with $r(\beta) = (\alpha, q)$ and $\delta(q, t(\alpha)) = \theta$, there is a set S satisfying θ, and for each $(a, q') \in S$, there is a direction b on r such that $r(\beta b) = (\alpha a, q')$.

Notice that there is no one-to-one correspondence beetween the nodes of r and t: a node of t could be represented several times (or none !) in r, with different labelling states: the loss of the ranked structure of t gives r the possibility to represent several different non-deterministic executions of a classical non-deterministic tree automaton at the same time.
One can thus understand a run-tree of an APT over a given tree as a new tree derivated from the previous one by possibly duplicating subtrees, and labelling every node of this new tree with a state, and the direction that corresponded to it in the former tree.

Example 4 (Run-tree of an APT)**.** We consider an APT on the ranked alphabet $\{g : 2\,; h : 1\,; a : 0\}$, with two states $\{1, 2\}$, and where the priority of a state is its name. The initial state is 1. We define δ as follows:

- $\delta(1, g) \;=\; (1, 1) \,\vee\, (2, 1)$
- $\delta(1, h) \;=\; (1, 1) \,\vee\, (1, 2)$
- $\delta(2, h) \;=\; (1, 2)$

and where δ is $\perp$ on other couples of $Q \times \Sigma$. Then the following trees are run-trees over $[\![G]\!]$ for G defined in Example 1[19]:

$$\begin{tikzcd}
 & & & g\,(1) \arrow[d, dash] & \\
 & g\,(1) \arrow[d, dash] & & g\,(1) \arrow[d, dash] & \\
g\,(1) \arrow[d, dash] & g\,(1) \arrow[d, dash] & & h\,(1) \arrow[dl, dash] \arrow[dr, dash] & \\
g\,(1) \arrow[d, dash] & h\,(1) \arrow[d, dash] & h\,(1) \arrow[d, dash] & & h\,(2) \arrow[d, dash] \\
h\,(1) \arrow[d, dash] & h\,(1) \arrow[d, dash] & h\,(2) \arrow[d, dash] & & h\,(2) \arrow[d, dash] \\
h\,(2) \arrow[d, dash] & h\,(2) \arrow[d, dash] & h\,(2) \arrow[d, dash] & & h\,(2) \arrow[d, dash] \\
h\,(2) \arrow[d, dash] & h\,(2) \arrow[d, dash] & h\,(2) \arrow[d, dash] & & h\,(2) \arrow[d, dash] \\
\vdots & \vdots & \cdots & & \vdots
\end{tikzcd}$$

All these run-trees are accepting: the only infinitely often occuring parity is 2, and it is even.
Note that $[\![G]\!]$ itself is not a run-tree of this APT; it could be one if we added the transition $\delta(1, a) \;=\; \top$. Without it, $[\![G]\!]$ can be labelled using δ, but δ does not hold at the a-labelled nodes, thus it is not a proper run-tree.

A run-tree r of an APT $\mathcal{A}$ over a tree t will then be considered as *accepting* iff it satisfies the classical parity condition, that is if on every infinite path over it, the smallest priority[20] met infinitely often is even.

[19]We represent the symbol of the original tree and the state labelling on the drawing.

[20]Recall that each node is labelled by a direction and a state; and every state has a priority given by Ω, which gives the node a priority.

APT and the modal μ-calculus model-checking over a tree. Emerson and Jutla proved in [3] the following result:

Theorem 2. *Checking whether a given modal μ-calculus formula holds at the root of a tree is equivalent to deciding whether there exists an accepting run-tree of some APT $\mathcal{B}$ over this tree.*

Emerson and Jutla's definition of APT being quite far from ours, one may refer to the section 2.3 of [11] for a formal translation. Here we focus only on the main ideas, and give some examples. The idea is that the purely Boolean subformulae of the formula ϕ to check at the root can be dealt with by the APT transition function directly[21]; the modalities can be easily dealt with too, $\diamond\psi$ corresponding to a transition with a *OR* over all the children of the current node, mentioning every possible direction, and with state affectations corresponding to verifying whether ψ holds on these children nodes. Similarly, $\Box\psi$ can be simulated with a *AND* over all directions, with a state labelling corresponding to checking ψ on these children nodes. Fixpoints are simulated using the parity function: the simulation with the APT corresponds to "unfolding" the formula, performing the recursion on the branches of the tree; the μ operators correspond to odd parity states and the ν ones to even parity states; and the outermost fixpoint operator has the smaller parity, the outermost ν operator being given the smallest parity.
Remember that the expected behaviour of a μ operator is to provide a *finite* recursion; if it does not, then the run-tree the model-checking APT labels has infinitely many states with the odd priority corresponding to this μ operator, and no infinite occurence of a smaller priority: this means that an operator which is less nested than this μ one occurs infinitely, and this contradicts the fact that this μ lead to an infinite recursion. So, the parity acceptance condition ensures that the only (possibly) infinitely recursive operators are the ν ones. Putting everything together, there is an accepting run-tree of the model-checking APT over a tree iff ϕ holds at its root.

Example 5 (Properties and APTs)**.** Now we consider $[\![G]\!]$ from Example 1, ϕ from Example 3, and the APT from Example 4. Remember that $\phi = \mu Z.(\psi \vee \diamond X)$, where $\psi = \nu Y.(h \wedge \diamond Y)$. The APT from Example 4 then checks whether ϕ holds at the root of the tree it runs on, in the sense that there is an accepting run-tree of this APT over some tree iff ϕ holds at the root of this tree, that is, if there is a path on this tree which, after a finite number of nodes, enters an infinite branch labelled $hhh\cdots$.
Actually, the state 1 stands for the checking of ϕ on a node - that is why it is initial

[21]Remember that the transition function of the APT gives Boolean positive formulae: so a negation will have to be replaced by a *OR* over every other possibility of choice over $\mathrm{Dir}(f) \times Q$.

- and the state 2 stands for the checking of ψ. When the current node is labelled g, we explore one (or two) of the children, thanks to

$\delta(1, g) = (1, 1) \vee (2, 1)$[22], in order to determine whether ϕ holds at one of them. It can only happen a finite number of times, else 1 is the smallest infinite priority and the run-tree is not accepting. On a, $\delta = \bot$ so that finite branches does not appear in a run-tree (if they did, we may have a finite accepting run-tree, which would not satisfy ϕ). When a h is reached, we either label its children 1 or 2 (or both, by duplicating the subtree - recall Example 4). We have to consider labelling 1, because we may encounter a h but then a g, $\cdots$: the first h we meet is not necessarily the beginning of the infinite branch we want to find. Now it should be clear that this APT performs model-checking of ϕ on trees; the existence of accepting run-trees of it over $[\![G]\!]$, as we have seen in Example 4, proves that $[\![G]\!]$ effectively has an infinite branch labelled only with h, reachable in a finite (just in 2, actually) number of steps.

Traversal trees. Therefore, the model-checking of a modal μ-calculus property over $[\![G]\!]$ is equivalent to the problem of the existence of an accepting run-tree of some alternating parity tree automata $\mathcal{B}$, modelling the property to check over $[\![G]\!]$. Recall that the path-traversal correspondence gives a one-to-one correspondence beetween maximal paths on $[\![G]\!]$ and maximal traversals over $\lambda(G)$: this extends to a one-to-one correspondence beetween *run-trees* of $\mathcal{B}$ over $[\![G]\!]$ and *traversal-trees* of $\mathcal{B}$ over $\lambda(G)$, where a traversal-tree corresponding to some run-tree is a tree whose maximal branches are exactly the maximal traversals over the run-tree[23].

Example 6 (Run-trees and traversal-trees)**.** Let's consider G and $\lambda(G)$ from Example 1, and an APT C on the ranked alphabet $\{g : 2 ; h : 1 ; a : 0\}$, with two states $\{0, 1\}$, and where the priority of a state is its name. The initial state is 1. We define δ as follows:

- $\delta(1, g) = (1, 1) \vee (2, 1)$
- $\delta(1, a) = \top$
- $\delta(0, a) = \top$

and obtain the following traversal-tree over $\lambda(G)$:

[22]Recall that the first value is the simulated direction, and the second one the labelling state.

[23]Note that, a traversal-tree being labelled by an alphabet wider than just Σ, the state labelling is performed by the APT on the children of Σ-nodes, and the value is then propagated along the traversal to the next Σ symbol.

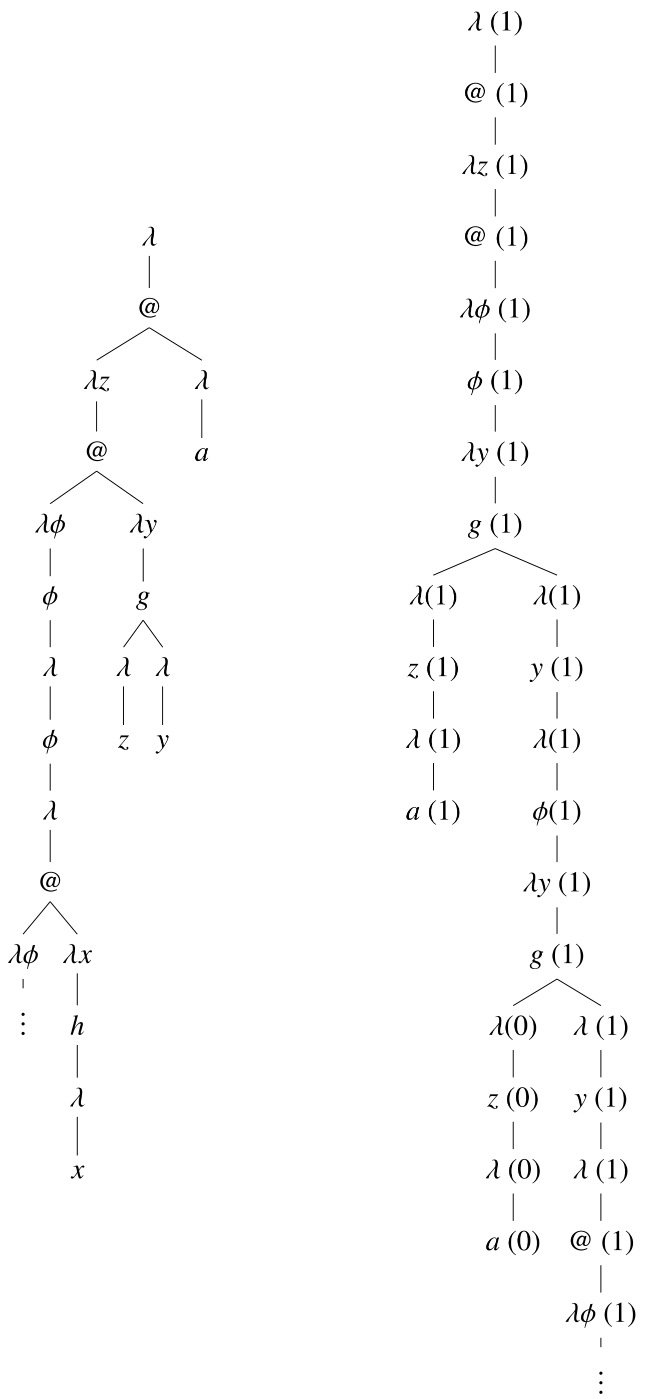


Figure 2: $[\![\lambda(G)]\!]$ and the corresponding traversal-tree for C

Thus, our model-checking problem is equivalent to the problem of the existence of an accepting traversal-tree over $\mathcal{B}$.

2.3 Informal extension of the process to the μ-modal calculus model-checking over graphs

From a given graph, as mentionned in Footnote 4, one can obtain a tree by *unravelling*: starting from a given root node of the graph, we explore each path on it incrementally; when a cycle is found, we replace the last visited edge by an edge to a *copy* of the subtree rooted at the node the removed edge was pointing to. Proceeding this way gives a tree which is equivalent to the graph in the sense that the set of paths over the graph and the tree obtained by unravelling are the same. Figure 10 in Section 4.1 gives an illustration of a graph and the tree obtained by unravelling it.
Then, the problem of deciding whether a given modal μ-calculus formula holds at a distinguished node of a graph is equivalent to the problem of deciding whether this formula stands at the root of the tree obtained by unravelling the graph starting at the distinguished node; and we have just seen how to solve this problem.

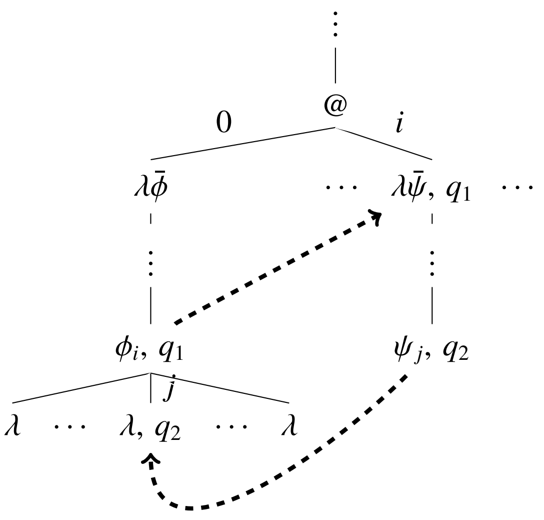
3 Simulating Traversals

3.1 On "model-checking run-trees“: the APT approach

Now that our model-checking task over $[\![G]\!]$ has been reduced to deciding whether there exists an accepting traversal-tree of $\mathcal{B}$ over $\lambda(G)$, we build an APT C whose accepting run-trees are accepting traversal-trees of $\mathcal{B}$.

Building from $\mathcal{B}$ a traversal-tree recognizing APT: an informal description. In order to simulate traversal-trees over the computation tree $\lambda(G)$ with an APT, we have to *simulate* properly the jumps in the tree, as an APT is essentially a top-down device. Thus, just labelling the run-trees with states of $\mathcal{B}$ does not suffice, and some more information needs to be added: this is the raison d'être of *variable profiles* and of *environments*, which are introduced formally in the next paragraph.
The idea of C is that when it arrives on an @ node, the automaton guesses the possible behaviours after computing the arguments (that is, the node a traversal jumps back to, the minimal priority encountered during the computation of the argument, and the state the computation of the argument ends with), and checks them when proceeding the non-0 directions (if there is a bad guess, the automaton

blocks), while the applicative code (in direction 0) uses the guessed values. Thus, in the following situation where ϕ is an order two variable, the expected behaviour of C is expected to be:



@
0
i
$\lambda\bar{\phi}$
$\cdots$ $\lambda\bar{\psi}$, q_1 $\cdots$
ϕ_i, q_1
ψ_j, q_2
j
λ $\cdots$ λ, q_2 $\cdots$ λ

- When C reaches the @ node, it *guesses* an *environment*, which is a set of "descriptions" like "computing the i^{th} argument may return to the j^{th} son of ϕ in state q_1, and the computation's least encountered priority is m_1 - or it may return to this j^{th} son in state q_2 and with minimal encountered priority m_2"
- On the branches describing arguments, the automaton checks that the environment it guessed at the @ node describes exactly all possibilities, and no more.
- On the applicative branch, when a variable is encountered, its sons are labelled with the states corresponding to the ones we guessed possibly encountered during the computation of the corresponding argument.

Variable profiles, active profiles, environments. Let us make all this a bit more formal. We need some definitions first:

Definition 8 (Variable profile)**.** A variable profile for $\mathcal{B}$ is a tuple (ϕ, q, m, c), where ϕ is the variable it describes (of type $A_1 \to \cdots \to A_n \to o$), q is a state of $\mathcal{B}$, m is a priority of $\mathcal{B}$, and c is a set of variable profiles for variables typed A_i, called its *interface*.

Definition 9 (Active profile)**.** An active profile for the APT $\mathcal{B}$ is a tuple $(\phi, q, m, c)^b$ which is a variable profile enriched with a boolean b which is false if the priority m was not encountered at the current step of the computation, and true if it was the lowest priority encountered so far - if a lower priority was found, then the automaton will be designed to block during the update of the boolean.

Definition 10 (Environment)**.** An environment ρ for the APT $\mathcal{B}$ over $\lambda(G)$ is a set of active profiles describing variables occuring as labels in $\lambda(G)$.

A *variable profile* is then describing some variable ϕ, giving the state q that is being simulated when ϕ is met by the automaton simulating the traversal, and the priority m that was the lowest encountered beetween the root and the last occurence of ϕ it meets[24], and an *interface* which describes the variable profiles that will label the children nodes of the ϕ-labelled nodes. An *active profile* adds a boolean whose use is to control that m is actually the smallest priority encountered so far. An environment gives such information for every occuring variable; it may have several profiles for a variable if its computation has several possible ways of returning when following a traversal.

Description of the behaviour of the traversal-simulating APT C. As we remarked previously, we need to label the run-trees of C with more than just states of $\mathcal{B}$ to make it simulate properly traversals: thus, states of C will be either of the shape $q\,\rho$ or $q\,\rho\,\theta$, where q is a $\mathcal{B}$-state, ρ is an environment for $\mathcal{B}$ over $\lambda(G)$, and θ is a distinguished variable profile. The starting state is then $q_0\ \emptyset$, and the priority of a state is the $\mathcal{B}$-one of q if there is no distinguished variable profile, and the one contained in the distinguished variable profile else. For the transition function, we just give a case-by-case description of the expected behaviour of C:

On applicative nodes, labelled $q\,\rho$: Remember the local structure of a computation tree: @ has $n + 1$ children, where n is the number of parameters of the application; the "code" of the application is in direction 0, starting with $\lambda\phi_1 \cdots \phi_n$, and each ϕ_i describes the i^{th} argument of the current application, so that its call-by-name β-reduction can be simulated by jumping to the child in direction i of @ when ϕ_i is reached in the applicative code which is in direction 0. First of all, C guesses a set $c = \{\theta_j = (\xi_{i_j}, q_{l_j}, m_j, c_j)\}$ of active profiles describing the variables in $\bar{\phi}$[25], and labels direction 0 with $q\ c$. For the other directions, recall that ρ corresponds to information on variables that were declared before; for each direction $i \geq 1$, the automaton

[24]That is, the last occurence of ϕ bound by the very same binder node.

[25]A given variable may be represented several times, corresponding to the different state labellings that may occur during a run of $\mathcal{B}$.

guesses which variables from ρ will be used, and labels the child in direction i_j with state q_{l_j}, environment the union of the profiles guessed useful from ρ, with booleans updated to *true* if the minimal priority they describe equals m_j (and blocking if the maximal priority of some of the active profiles is strictly bigger than the one of q_{l_j}), and of c_j updated similarly with respect to priority $\Omega(q_{l_j})$, and with distinguished profile θ_j.

On λ nodes, labelled $q\ \rho$ or $q\ \rho\ \theta$: Such a node only has one child; which the APT labels $q\ \rho\ \tau$ (with $\tau^b \in \rho$ a profile for some variable ψ) if it guesses that this child is labelled by a variable ψ; else it labels its child $q\ \rho$.

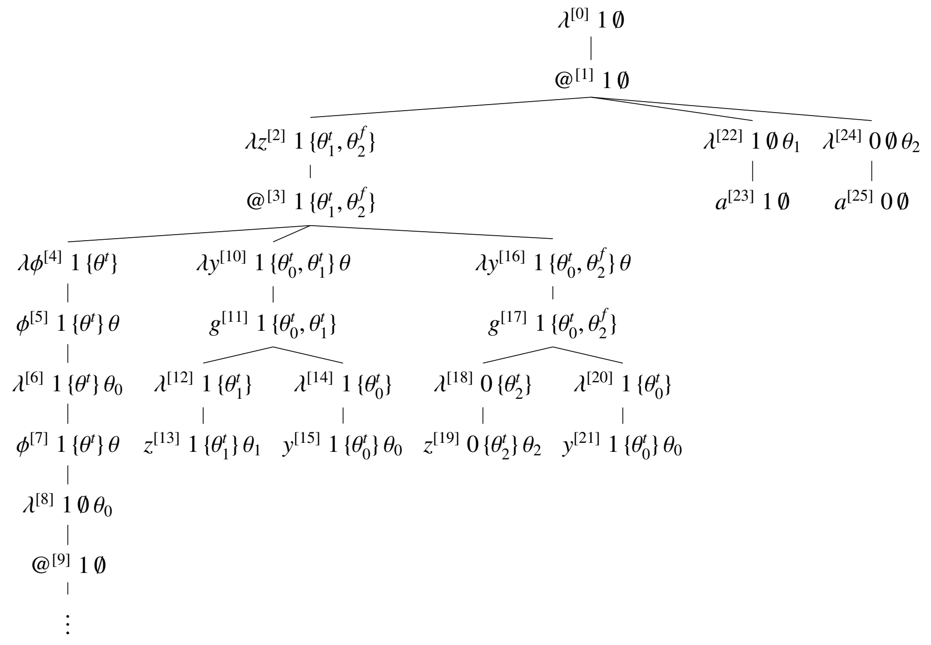
On Σ nodes, labelled $q\ \rho$: On such a node, the APT first has to guess a set of directions and states $\{(i_1, q_{j_1}), \cdots, (i_k, q_{j_k})\}$ satisfying $\delta(q, f)$, where δ is the transition function of the property APT $\mathcal{B}$, in order to fulfill its verificational purpose. Then, the automaton guesses which active profiles from ρ will be used in a given direction i_l, and labels the direction i_l with the state q_{j_l} and the environment made from the union of the active profiles of ρ guessed useful for direction i_l, after updating their booleans to *true* if the minimal priority they describe equals $\Omega(q_{j_l})$ (and blocking if the maximal priority of some of the active profiles is strictly bigger than the one of q_{j_l}). Note that every active profile from ρ has to be allocated at least to one direction, as we want C to always guess the smallest possible environment.

On variable nodes, labelled $q\ \rho\ \theta$: On a variable node ϕ, the immediatly previous λ-node should have distinguished as $\theta = (\phi, q, m, c)$ a profile for ϕ, with $\theta^t \in \rho$ (else C blocks), describing the result of the β-reduction jump, $c = \{\theta_j = (\xi_{i_j}, q_{l_j}, m_j, c_j)\}$ describing the active profiles to assign as labels to the children of ϕ (so that $c = \emptyset$ iff ϕ has order 0) - there is nothing to check here as the check only takes place in the other directions. First of all, the automaton guesses whether ϕ will be used again before it eventually gets re-bound in another context[26]; define ρ' to be ρ if it is the case, and $\rho \setminus \theta$ if it is not - remember that an environment should only contain useful data, else the automaton blocks at some point. Then, for each direction i_j, the automaton guesses which profiles from ρ' will be used - each profile being assigned at least to one direction - and labels the child in direction i_j with state q_{l_j}, environment the union of guessed useful profiles updated to priority m_j and of profiles of c_j updated to $\Omega(q_{l_j})$, and distinguished variable θ_j.

[26]Recall that the recursion schemes have a recursive power, thus the same variable name may occur infinitely many times, but refering to infinitely many different binders (which can be identified by their justification links), because there may be an infinite recursion on the same application.

Let us see how this behaves on an example:

Example 7 (A run-tree of C)**.** We consider the APT of Example 6, over $\lambda(G)$ of Example 1. We denote $\theta = (\phi, 1, 1, \{\theta_0\})$, $\theta_0 = (y, 1, 1, \emptyset)$, $\theta_1 = (z, 1, 1, \emptyset)$, $\theta_2 = (z, 0, 0, \emptyset)$. Then the following tree is a run-tree for this APT over $\lambda(G)$ (we added numbers as superscripts to refer easily to a given node):


$\lambda^{[0]}\ 1\,\emptyset$
$@^{[1]}\ 1\,\emptyset$
$\lambda z^{[2]}\ 1\,\{\theta_1^t, \theta_2^f\}$
$\lambda^{[22]}\ 1\,\emptyset\,\theta_1$
$\lambda^{[24]}\ 0\,\emptyset\,\theta_2$
$@^{[3]}\ 1\,\{\theta_1^t, \theta_2^f\}$
$a^{[23]}\ 1\,\emptyset$
$a^{[25]}\ 0\,\emptyset$
$\lambda\phi^{[4]}\ 1\,\{\theta^t\}$
$\lambda y^{[10]}\ 1\,\{\theta_0^t, \theta_1^t\}\,\theta$
$\lambda y^{[16]}\ 1\,\{\theta_0^t, \theta_2^f\}\,\theta$
$\phi^{[5]}\ 1\,\{\theta^t\}\,\theta$
$g^{[11]}\ 1\,\{\theta_0^t, \theta_1^t\}$
$g^{[17]}\ 1\,\{\theta_0^t, \theta_2^f\}$
$\lambda^{[6]}\ 1\,\{\theta^t\}\,\theta_0$
$\lambda^{[12]}\ 1\,\{\theta_1^t\}$
$\lambda^{[14]}\ 1\,\{\theta_0^t\}$
$\lambda^{[18]}\ 0\,\{\theta_2^t\}$
$\lambda^{[20]}\ 1\,\{\theta_0^t\}$
$\phi^{[7]}\ 1\,\{\theta^t\}\,\theta$
$z^{[13]}\ 1\,\{\theta_1^t\}\,\theta_1$
$y^{[15]}\ 1\,\{\theta_0^t\}\,\theta_0$
$z^{[19]}\ 0\,\{\theta_2^t\}\,\theta_2$
$y^{[21]}\ 1\,\{\theta_0^t\}\,\theta_0$
$\lambda^{[8]}\ 1\,\emptyset\,\theta_0$
$@^{[9]}\ 1\,\emptyset$
$\vdots$

- At the beginning, the node [0] is labelled with the initial state $1\ \emptyset$. The root is a λ, its child is not a variable: so [1] is labelled $1\ \emptyset$.

- [1] is an applicative node. Thus it guesses that z will be used, and its evaluation in a run-tree of $\mathcal{B}$ may either return in state 1 or 0, with respective smallest encountered priority 1 or 0; it also guesses that both possibilities will be used in the current run: so c is set equal to $\{\theta_1^t, \theta_2^f\}$, and each of these two possible values is explored, respectively in direction 1 and 2, with an empty environment (because ρ and the interfaces of θ_1 and θ_2 are), with distinguished profile θ_1 or θ_2.

- In directions 1 and 2, λ guesses that a is not a variable node, thus labels its child with $1\ \emptyset$; then on a the automaton stops the exploration, but does not block as $\delta(1, a) = \delta(2, a) = \top$.

- In direction 0, the node [2] is met; it is a λ, which guesses that its child is not a variable and thus does not distinguish a variable profile from ρ for it.

- At the node [3], C first guesses an active profile[27] θ for ϕ, which is the new introduced variable, refering the only argument of this new application. θ embeds, as its interface, the information that its evaluation results (in the current run) to labelling its child with θ_0; to verify this guess, and provide information about the evaluation of z, C affects the environments $\{\theta_0^t, \theta_1^t\}$ and $\{\theta_0^t, \theta_2^f\}$ to the directions 1 and 2 - again, the two possibilities are explorated simultaneously in this run, by duplicating the subtree rooted on [10].

- In direction 1, the λ-node [10] guesses that its son is not a variable, and thus does not provide it a distinguished profile. Then g dispatches the profiles of its environment to its two children, the λ nodes then provide to the order-0 variables at nodes [13] and [15] a distinguished profile that matches them; the automaton stops the exploration in these directions but does not block. In direction 2, a similar process occurs.

- In direction 0, C arrives at node [4], which is a λ. It guesses that its child is ϕ, and thus distinguishes a profile for ϕ - here there is no choice, so θ is chosen.

- C goes to node [5]: it is a variable of order non-0, with only one child. C guesses that ϕ will be used again within the range of its binder (node [4]), and thus does not suppress its profile from the environment. θ_0 is affected as a distinguished label to the node [6], and state is still 1, as θ_0, which represents the β-reduction process of ϕ, indicates that this process returns in state 1.

- At λ-node [6], C guesses that its child is ϕ, and thus provides it a suitable distinguished profile.

- [7] agains requires a β-computation of ϕ, which (in this run) behaves as θ_0 describes it. C guesses that ϕ will not be used anymore, so it is suppressed from the environment that is puts to [8] as a label; θ_0 is added as a distinguished profile in [8] as the return value of the β-reduction.

- At λ-node [8], C guesses that its child is not a variable, and does not distinguish a profile in the labelling it gives it.

- At node [9], a new application starts; so does a similar process.

[27]There may have been several, if the β-reduction path that is being simulated lead to different return states; in the run presented in this example, this is not the case.

We then have that C simulates the traversals of $\mathcal{B}$, in the sense that there is an accepting run-tree of C over $\lambda(G)$ iff there is an accepting traversal-tree of $\mathcal{B}$ over $\lambda(G)$.[28]

3.2 A machine equivalent to the *n*-RS: the *n*-CPDA

Another way of simulating traversals of $\lambda(G)$ was introduced in [5], and is based on a new kind of machine, called (n-)Collapsible Pushdown Automata, whose expressivity is the same as (order-n) recursion schemes, and whose behaviour allows a proper simulation of traversals over the computation tree of a recursion scheme. We first introduce these new machines, and then explain how they simulate traversals over $\lambda(G)$.

***n*-CPDA.** n-CPDA are essentially pushdown automata working on stacks of stacks of stacks . . . of symbols enriched with links to stacks of lower order situated below them in the n-stack; a 1-CPDA has an usual stack plus links, a 2-CPDA has a stack of stacks, . . . These devices generate trees.

Definition 11 (n-stacks with links)**.** A 0-stack is just a stack alphabet symbol; a n-stack is a sequence of $(n-1)$-stacks where non-$\perp$ symbols have a link to a previous stack (of any order). $\perp$ is a distinguished stack bottom symbol that should be the bottom symbol of any 1-stack occuring in the n-stack, and not appear at any other position.

Definition 12 (n-stack operations)**.** The operations on a n-stack are the following ones:

top_i returns the top $(i-1)$-stack of the argument n-stack.

pop_i removes the top $(i-1)$-stack from the argument n-stack.

Order 1 *push***:** $push_1^{a,k}$ adds a to the top position of the top 1-stack, with a link to the closest $(k-1)$-stack.

Order $i \geq 2$ *push* just duplicates the top $(i-1)$-stack; it preserves the link structure, in the sense that links that originally pointed outside of the top $(i-1)$-stack are duplicated as links pointing exactly to the same position, and in the duplicated stack links that were pointing inside the top $(i-1)$-stack points to their duplicated equivalent.

Collapse removes everything that is over the position the top symbol points to.

[28] See [9], sections 4 and 5, for a proof.

Example 8 (Some stack operations)**.** Start with the 3-stack $s = [[[\bot a]][[\bot][\bot a]]]$ (we do not represent links which points to the immediate previous symbol, for the sake of readability). We perform on it several operations:

- $push_1^{b,2}\, s = s_1 = [[[\bot a]][[\bot][\bot a b]]]$
- $push_1^{c,3}\, s_1 = s_2 = [[[\bot a]][[\bot][\bot a b c]]]$
- $push_2\, s_2 = s_3 = [[[\bot a]][[\bot][\bot a b c][\bot a b c]]]$
- $push_3\, s_2 = s_3' = [[[\bot a]][[\bot][\bot a b c]][[\bot][\bot a b c]]]$ [29]
- $collapse\, s_3 = collapse\, s_3' = [[[\bot a]]]$

Definition 13 (n-CPDA)**.** A n-CPDA is a tuple $\mathcal{A} = <\Sigma, \Gamma, Q, \delta, q_0>$ where Σ is a ranked output alphabet, Γ is the stack alphabet (with a stack bottom symbol $\bot$), Q is a finite state set, q_0 is the initial state, and $\delta : Q \times \Gamma \to Q \times Op_n \cup \{(f; q_1 \cdots q_{ar(f)}) / f \in \Sigma, q_i \in Q\}$[30] maps a state and the stack top symbol to either an operation to perform on the stack and a new state, either to an output move.
Transitions are defined over *configurations* (q, s) with $q \in Q$ and s a n-stack; the initial configuration is $(q_0, \bot_n)$ where $\bot_n$ is the empty n-stack $[\cdots[\bot]\cdots]$. There are three kinds of transitions:

- Internal moves: $(q, s) \xrightarrow{(q',\theta)} (q', s')$ if $\delta(q, top_1\, s) = (q', \theta)$ and $s' = \theta(s)$
- Proponent moves: $(q, s) \xrightarrow{(f, q_1, \cdots, q_{ar(f)})} (f; q_1, \cdots, q_{ar(f)}; s)$ if $\delta(q, top_1\, s) = (f, q_1, \cdots, q_{ar(f)})$
- Opponent moves: $(f, q_1, \cdots, q_{ar(f)}; s) \xrightarrow{(f,i)} (q_i, s)$ for each $1 \leq i \leq ar(f)$.

A *computation path* is then a sequence of configurations and transitions linking them, starting from the initial configuration.
One may recognize the vocabulary from *game semantics*[31]; the idea, as in the

[29] Notice here the difference of link structure beetween s_3 and s_3': the links inside the duplicated n-stack are duplicated so that they point to the copies, whereas links pointing outside of it are duplicated to point to the same stack as the original did.

[30] δ is required not to push or pop $\bot$ symbols.

[31] For an introduction, one may consult [1]

computation of traversals, is that the Proponent provides a choice of branches to explore, and the Opponent picks one; the internal moves are hidden in the result, as they just are local computations used to determine the symbols to output. We will use *deterministic* n-CPDA, in the sense that two transitions with the same label from a same configuration give the same result. In a sense, the idea is that the only "non-determinism" is not really one, but is a *choice* by the Opponent of a direction to explore. The *tree* generated by a n-CPDA is then the one that can be built from the computation of all its branches, that are all the possible *computation paths* of the n-CPDA.

n-CPDA simulate traversals. n-CPDA and order-n recursion schemes are equi-expressive: they generate the same trees. The idea[32] is that one can encode the n-stacks as properly typed non-terminals of a recursion scheme, and the converse inclusion comes from the fact that traversals over the computation tree $\lambda(G)$ of some recursion scheme can be computed by a traversal-simulating n-CPDA; then it follows from the Path-Traversal correspondence that this n-CPDA computes every path (actually, the one the Opponent requires by his successive choices) of $[\![G]\!]$. In the following, we sketch this converse inclusion, with an emphasize on the simulation of traversals by n-CPDAs.
In the following, we denote $E_i(u)$ the child in direction i of the node u of $\lambda(G)$, and v_0 the root of $\lambda(G)$.

Definition 14 (The traversal-computing n-CPDA $\mathcal{A}$)**.** For a fixed order-n recursion scheme G, we define its *traversal-computing* n-CPDA to be the automaton whose output alphabet is the alphabet used for the output of G, whose stack alphabet is the set of labels of nodes of $\lambda(G)$, and whose initial configuration is $(q_0, [\cdots[\perp v_0]\cdots])$[33], and whose transition function acts as follows, depending on the top stack symbol u[34]:

- If u is the label @ of some applicative node, $\delta(u) = push_1^{E_0(u)}$.
- If u is the label f of some Σ-node, $\delta(u) = push_1^{E_i(u)}$, where $1 \leq i \leq ar(f)$ is the direction the Opponent chooses.[35]

[32]Proofs in [5].

[33]This is not exactly the definition of n-CPDA we used, but it should be immediate to see that adding an initial state that pushes v_0 and goes to state q_0 gives an equivalent automaton fulfilling the definition.

[34]Instead of giving a formal definition, we give here a definition which allows to compose several n-stack operations - a formal definition would need to consider intermediate states therefore. Thus we do not consider states explicitely here.

[35]This case is the one generating trees: this transition hides the output of f by the Proponent, plus the choice of a direction to compute by the Opponent. All the other cases correspond to internal moves.

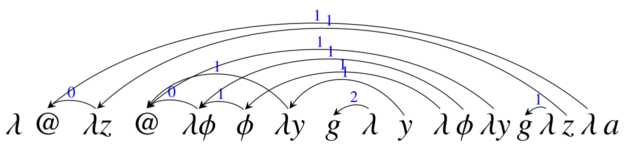
- If u is a λ-node, $\delta(u) = push_1^{E_1(u)}$.

- If u is a variable ϕ of order l, which is the i^{th} parameter of its binder; that this binder is a j[36]-child, and that there are p nodes on the path in $\lambda(G)$ from ϕ to its binder, comprising these two nodes, we have:

  - If $l \geq 1$ and $j = 0$, $\delta(u) \; = \; push_{n-l+1} \, ; \, pop_1^p \, ; \, push_1^{E_i(top_1),n-l+1}$ (sequential composition of operations on the n-stack being denoted by ;, and the power refering to the iteration of this sequential composition)
  - If $l, j \geq 1$, $\delta(u) \; = \; push_{n-l+1} \, ; \, pop_1^{p-1} \, ; \, collapse \, ; \, push_1^{E_i(top_1),n-l+1}$
  - If $l = 0$ and $j = 0$, $\delta(u) \; = \; pop_1^p \, ; \, push_1^{E_i(top_1)}$
  - If $l = 0$ and $j \geq 1$, $\delta(u) \; = \; pop_1^{p-1} \, ; \, collapse \, ; \, push_1^{E_i(top_1)}$

We first consider a run on an example:

Example 9 (Runs and traversals)**.** Here we consider the traversal-computing 2-CPDA associated to $\lambda(G)$ of Example 1. Recall that the following justified sequence over $\lambda(G)$ is a traversal, as seen in Example 2:


$\lambda \; @ \; \lambda z \; @ \; \lambda\phi \; \phi \; \lambda y \; g \; \lambda \; y \; \lambda\phi \; \lambda y \; g \; \lambda z \; \lambda \; a$
0
1
1
1
1
1
0
1
1
2
1

We represent here the stack configurations that occur during the computation of this traversal by the traversal-computing 2-CPDA. Trivial steps may be skipped (denoted by the use of $\rightarrow^*$), and trivial links are not displayed.

[36]So, $j = 0$ means that the variable corresponds to an actual parameter of the application; $j \geq 1$ means that the the variable is a local one, used while proceeding a β-reduction.

$$
\begin{array}{lll}
(1) & & [\,[\,\bot\ \lambda\,]\,] \\
(2) & \to^{*} & [\,[\,\bot\ \lambda\ \ @\ \lambda z\ @\ \lambda\phi\ \phi\,]\,]
\end{array}
$$

$$\begin{tikzcd}[column sep=small] (3) & \to & {[\,[\,\bot\ \lambda\ \ @\ \lambda z\ @\ \lambda\phi\ \phi\,]} & {[\,\bot\ \lambda\ @\ \lambda z\ @} & \lambda y \arrow[ll, bend right] & {]\,]} \end{tikzcd}$$

$$\begin{tikzcd}[column sep=small] (4) & \to^{*} & {[\,[\,\bot\ \lambda\ @\ \lambda z\ @\ \lambda\phi\ \phi\,]} & {[\,\bot\ \lambda\ @\ \lambda z\ @} & \lambda y \arrow[ll, bend right] & {g\ \lambda\ y\,]\,]} \end{tikzcd}$$

$$
\begin{array}{lll}
(5) & \to & [\,[\,\bot\ \lambda\ @\ \lambda z\ @\ \lambda\phi\ \phi\ \lambda\,]\,] \\
(6) & \to & [\,[\,\bot\ \lambda\ @\ \lambda z\ @\ \lambda\phi\ \phi\ \lambda\ \phi\,]\,]
\end{array}
$$

$$\begin{tikzcd}[column sep=small] (7) & \to & {[\,[\,\bot\ \lambda\ @\ \lambda z\ @\ \lambda\phi\ \phi\ \lambda\ \phi\,]} & {[\,\bot\ \lambda\ @\ \lambda z\ @} & \lambda y \arrow[ll, bend right] & {]\,]} \end{tikzcd}$$

$$\begin{tikzcd}[column sep=small] (8) & \to^{*} & {[\,[\,\bot\ \lambda\ @\ \lambda z\ @\ \lambda\phi\ \phi\ \lambda\ \phi\,]} & {[\,\bot\ \lambda\ @\ \lambda z\ @} & \lambda y \arrow[ll, bend right] & {g\ \lambda\ z\,]\,]} \end{tikzcd}$$

$$
\begin{array}{lll}
(9) & \to & [\,[\,\bot\ \lambda\ @\ \lambda z\ @\ \lambda\phi\ \phi\ \lambda\ \phi\,]\,[\,\bot\ \lambda\ @\ \lambda\,]\,] \\
(10) & \to & [\,[\,\bot\ \lambda\ @\ \lambda z\ @\ \lambda\phi\ \phi\ \lambda\ \phi\,]\,[\,\bot\ \lambda\ @\ \lambda\ a\,]\,]
\end{array}
$$

- From step 1 to step 2, the automaton just goes down in the tree, visiting the 0-child when it finds a @[37], starting from the root, and it stores what it finds.

- From step 2 to step 3: the top_1 stack value was a variable node of order $l = 1$, and we have $j = 0$ because ϕ refers to an actual argument of the application. Thus the first rule for variables applies: the automaton duplicates the top 1-stack with a $push_2$ operation, then removes every top_1 symbol until it finds the @ of the application which ϕ refers to, then it pushes its child in direction 1 (so, the code corresponding to the first argument, which corresponds to ϕ), with a link to the former 1-stack so that collapsing from there returns to where the β-jump started.

- From step 3 to step 4: the exploration of the argument's code is performed; the Opponent chooses to explore the direction 2 when g is visited.

- From step 4 to step 5: the top_1 symbol is an order-0 variable, thus the current β-reduction is finished. For y, $j = 1$, as it corresponds to the first (and only) argument of ϕ. Thus, the traversal jumps back to where the β-jump started, by successive *pop*s until the binder of y and then a *collapse* that returns; and the traversal now knows that it has to explore the child of ϕ in direction 1: so it pushes the corresponding label (the rule here is the fourth for variables).

[37]Recall a traversal simulates a *call-by-name* β-evaluation of a program, so in an application, the applicative code is visited first, and arguments only are when they are refered to.

- From step 5 to step 6: the exploration continues; ϕ is found.

- From step 6 to step 7: here the behaviour is similar to what happened from step 2 to step 3.

- From step 7 to step 8: the exploration of the argument's code is performed; the Opponent chooses to explore the direction 1 when g is visited.

- From step 8 to step 9: the top_1 symbol is a z, which is an order-0 variable: the current β-jump is then finished, and should return to the argument corresponding to z, which is the first argument of the first application the traversal met while proceeding the tree. The rule here is the third for variables, which leads the automaton to *pop* everything until the @ node z is associated with (so, the first in the tree) is found; then its first child, whose value corresponds to the one z represents, is visited and pushed on the top.

- From step 9 to step 10: the exploration continues; a is found. As its arity is 0, the Opponent has no move and the computation stops. One may recognize that we simulated here the traversal corresponding to the branch $g \cdot g \cdot a$ in $[\![G]\!]$.

In this example, the 2^{nd} rule for variables was not used: it is only used at order at least 3, when the depth of β-jumps may be more than just 1. One may refer to [4, Appendix A] for an (order-3) example. The idea is that, at higher orders, the computation of an argument during a β-reduction (of, say, ϕ) can itself need a β-reduction (of, say, ξ). Then, the current context is saved by performing a $push_{n-l+1}$ operation: notice that, l being smaller for the variables occuring inside a β-jump, this duplication includes a "save" of the current state of the β-jump (computing ϕ) which is requiring to launch this new β-jump (computing ξ). Then, we need to access the subtree which stores the code for the argument that corresponds to ξ; and ξ itself describes one argument of ϕ. The traversal thus has to visit the corresponding child of ϕ: for this, after the copy made by the $push_{n-l+1}$ operation, we *pop* until the binder of ξ is the top_1 symbol and then collapse, going back this way to ϕ; then we can visit the subtree that corresponds to the argument of ϕ described by ξ by pushing the son in appropriate direction of the current node (ϕ).

Traversals and n-stacks configurations One may have noticed that reading the 2-stacks of the example does not exactly simulates the growth of the traversal. There is a little transformation to perform:

- On n-stacks: if s is a n-stack, we define the sequence $\bar{s}$ which is obtained by starting to read the n-stack from the right, ignoring every [,] , or $\perp$ symbol, and following links (thus $\bar{s}$ does not contain the nodes beetween the source and the destination of a link, these two being included in $\bar{s}$). In Example 9, one may check that at step 8, $\bar{s} = @\ \lambda z\ @\ \lambda\phi\ \phi\ \lambda\ \phi\ \lambda y\ g\ \lambda\ z$

- On traversals: if t is a traversal, we define $\hat{t}$, obtained from t by removing every w sandwiched between matching pairs of the shape $\overset{\overset{i}{\curvearrowleft}}{\$\ \ \lambda}$, with $i \geq 1$ and $\$$ an order-1 variable or an @ symbol (so that $w = \overset{\overset{i}{\curvearrowleft}}{\lambda\bar{\phi} \cdots \phi_i}$ where ϕ_i has order 0). The information stored in $\hat{t}$ suffices to define the traversal it represents: hiding the internal moves corresponding to the β-reduction of order-1 variables does not matter, as the node where this computation returns, which describes the result of this β-reduction, is kept in $\hat{t}$.

We then define the computation of a traversal by a n-stack (of a n-CPDA) as follows:

Definition 15. If s is a reachable n-stack of the traversal-computing n-CPDA of G, and t a traversal over $\lambda(G)$, s is said to compute t iff all of the following holds:

- $top_2\, s = \lceil t \rceil$
- $\bar{s} = \hat{t}$
- If $top_2\, s = [\perp s_1 \cdots s_n]$, and $v_1, \cdots, v_n$ are the labels of t that corresponds to $s_1, \cdots, s_n$, then $pop_1^{n-i}\, s$ computes the prefix of t ending at v_i, included.
- If $top_2\, s = [\perp s_1 \cdots s_n]$, and $v_1, \cdots, v_n$ are the labels of t that corresponds to $s_1, \cdots, s_n$, then $collapse(pop_1^{n-i}\, s)$ computes the prefix of t ending at v_i, excluded.

The result is then that:

Theorem 3. *If s computes t, then s and t are bisimilar, in the sense that:*

- *If s' is reachable from s, then t can be extended to t' such that t is a prefix of t' and t' is a traversal over $\lambda(G)$, and s' computes t'.*
- *If t can be extended to t' such that t is a prefix of t' and t' is a traversal over $\lambda(G)$, then there is an n-stack s' reachable from s which computes t'.*

Then, n-CPDA can compute traversals, as the APT we built in the previous section could. Both devices may then be used for a traversal-based resolution of our model-checking problem.

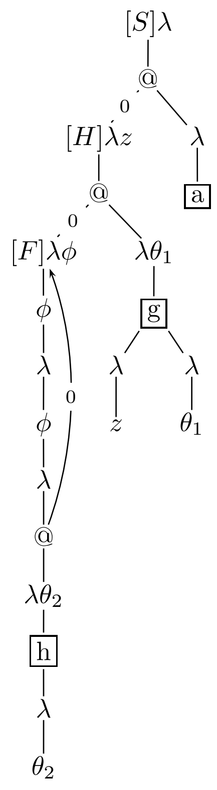
4 Solving the parity games

Now that we obtained *traversal-simulating* devices, we may perform verification using them. For the APT approach, we first build a parity game whose resolution is equivalent to the existence of an accepting run-tree of the APT over a given tree: we can then use it for C over $\lambda(G)$ and perform our verification task this way. For the n-CPDA approach, we use a similar kind of verification parity game, over the set of its configurations - provided that n-stacks of a traversal-computing n-CPDA compute the traversals over a given tree, this gives a way to model-check these traversals over the computation tree of a recursion scheme, and thus the tree generated by the scheme thanks to the Path-Traversal correspondence.

4.1 On the APT

An APT defines a parity game. As seen in Section 2, a given run-tree is *accepting* iff it satisfies the usual parity condition on parity games; we shall see here that there is a way, given an APT C and a tree $\lambda(G)$, to build a parity game where Eloïse has a winning strategy iff there is an accepting run-tree of C over $\lambda(G)$. The idea is that Eloïse has control on the guess of satisfying sets, so she chooses the run-tree to prove accepting; and Abélard chooses the path whose parity he wants to check.
To obtain a *finite* game, we need to turn $\lambda(G)$ into an "equivalent" graph, in the sense that unravelling it gives $\lambda(G)$. We gave an idea of it in footnote 4: instead of iterating *ad infinitum* the expansion of the rules of $[\![G]\!]$ to obtain a possibly infinite term and thus a possibly infite tree, we generate first a forest where every right part of a rule is a tree; then, we identify in this forest the nodes labelled by a non-terminal with the root of the tree which corresponds to this non-terminal in the initial forest. So we inserted *loops* in the generated graph, instead of iterating them. We denote this graph as $\mathrm{Gr}(G)$.

Example 10 (A generated graph.)**.** The RHS-transformed scheme from Example 1 leads to the following generated graph, where $[S]$, $[F]$ and $[H]$ indicate the roots of the corresponding rules:



From Gr(G) and C, we now build an acceptance parity game as follows: with Q the set of states of C and δ its transition function, we first define, for a given node v of Gr(G), and a set P of couples of possible directions for children of v and of states of Q, the set $[P]_v = \{(u, q) \,/\, (i, q) \in P$ and u is the child node of v in direction $i\}$. So, given a set P of possible labellings of directions by states[38] for the children of v, $[P]_v$ gives the set of every nodes with their state labelling that may appear in a run-tree as children of v. We can now define the acceptance parity game of C over Gr(G) as:

Definition 16 (Acceptance parity game of C over Gr(G))**.** The acceptance parity game of C over Gr(G) has:

- Two kind of vertices:
  - The vertices controlled by Abélard are the sets $[P]_v$, with v a node of Gr(G) and P a set of couples of direction for children of v and states for labelling them.

[38] P will be used as a *satisfying set* for δ: so $[P]_v$ gives the set of all possible children nodes with all possible state affectations for them in a run-tree of C.

  - The vertices controlled by Eloïse are the pairs of vertices of Gr(G) and states of Q.

- Two kind of edges:

  - From every Abélard-owned vertex $[P]_v$, there is for each $(u, q) \in [P]_v$ an edge from this vertex to the Eloïse-owned vertex (u, q).
  - From every Eloïse-owned vertex (v, q), there is for each set P of couples of directions and states satisfying $\delta(q, v)$ an edge from this vertex to the Abélard-owned vertex $[P]_v$.

- A priority function Ω which maps the vertices (v, q) to the priority of q given by C, and vertices $[P]_v = \{(u_1, q_1), \cdots, (u_r, q_r)\}$ to the maximal C-priority that is found among the C-priorities of $q_1, \cdots, q_r$.

A play is then a path on the parity game, starting from (v_0, q_0), where v_0 is the root of Gr(G), corresponding to the axiom S of G, and q_0 is the initial state of C; the play alternates between Eloïse-owned and Abélard-owned vertices. From an Eloïse-owned vertex, Eloïse chooses an Abélard node linked to the current node by an edge; Abélard acts similarly. If a play is finite, the player who cannot move looses; if a play is infinite, Eloïse wins iff the minimal priority among the ones which occur infinitely often is even.
It should be clear that Eloïse has a winning strategy for this game iff C has an accepting run-tree over Gr(G), and thus over $\lambda(G)$ by unravelling: the plays of Eloïse correspond to choosing locally which satisfying set for δ at the current node and in the current state should be used, so that Eloïse has the power to choose the run-tree she thinks is accepting, and Abélard chooses which branch of this run-tree should be explored; he tries to win, and thus he tries to find a branch which does not satisfy the parity condition. If there is no such one, Eloïse wins: that means that she provided an accepting run-tree of C over Gr(G). If Eloïse does not have any way to win, then it means that there is no such tree.

Complexity result. Now that we have a verification parity game for our modal μ-calculus model-checking problem, we may use the following theorem from [6]:

Theorem 4 (Jurdsziński 2000)**.** *The winning region of Eloïse and her winning strategy in a parity game with n vertices and m edges and $p \geq 2$ priorities can be computed in time:*

$$O(p \cdot m \cdot (\frac{n}{\lfloor \frac{p}{2} \rfloor})^{\lfloor \frac{p}{2} \rfloor})$$

This leads to:

Theorem 5 (Ong 2006)**.** *The modal μ-calculus model-checking problem for trees generated by order-n recursion schemes is n-EXPTIME complete, for every $n \geq 0$.*

because we have equivalence of the following assertions:

- A modal μ-calculus formula ϕ holds at the root of $[\![G]\!]$ generated by G
- Property APT $\mathcal{B}$ has an accepting run-tree over $[\![G]\!]$
- Property APT $\mathcal{B}$ has an accepting traversal-tree over $\lambda(G)$
- Traversal-simulating APT C has an accepting run-tree over $\lambda(G)$
- Eloïse has a winning strategy in the acceptance parity game for C over $\lambda(G)$

4.2 On the n-CPDS

Using the n-CPDA for computing traversals also leads to a model-checking procedure by solving parity games on the configuration graph (and thus on n-stacks of the traversal-computing n-CPDA, which, as we have seen, have a strong link with traversals). In this approach though, the verification is not performed on the generated tree nor on its computation tree $\lambda(G)$, but on the *configuration graph* of the automaton. This is equivalent, since n-CPDA and order-n recursion schemes are equi-expressive; informally the idea is that instead of working on the traversals of $\lambda(G)$ for our verification purpose, as in the APT approach, we may just work on the n-stacks of a traversal-computing n-CPDA given that they compute traversals. In the following, we will then focus on the verification of a given modal μ-calculus formula ϕ over a configuration graph of a slightly different device, the n-Collapsible Pushdown System (n-CPDS), which basically is a n-CPDA without output moves - the direction to compute will then be choosen by the Opponent in a (parity) game over the n-CPDS configuration graph.

n-CPDS and their configuration graphs. We define n-CPDS as follows:

Definition 17 (n-CPDS)**.** A n-CPDS is a tuple $\mathcal{A} = < \Gamma, Q, \delta, q_0 >$ where Γ is the stack alphabet (with a stack bottom symbol $\perp$), Q is a finite state set, q_0 is the initial state, and $\Delta : Q \times \Gamma \times Q \times Op_n$.[39]
Transitions are defined over *configurations* (q, s) with $q \in Q$ and s a n-stack; the initial configuration is $(q_0, \perp_n)$ where $\perp_n$ is the empty n-stack $[\cdots[\perp]\cdots]$. We removed output moves, so the only kind of transitions is now the one corresponding

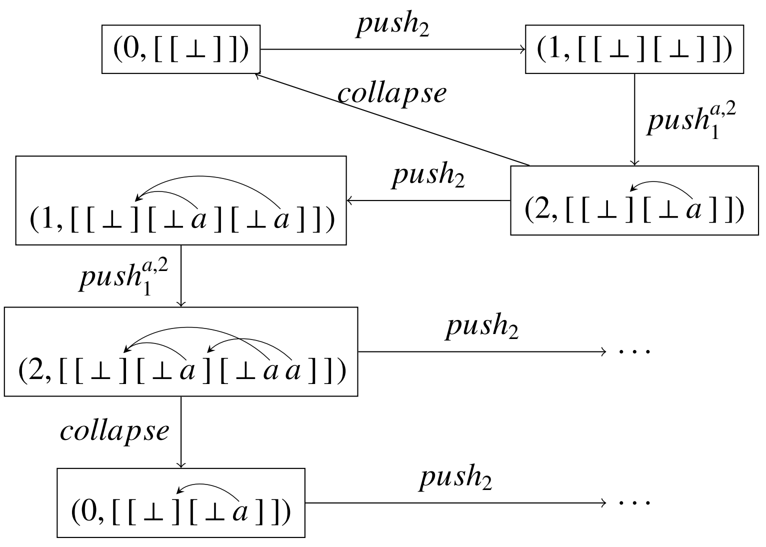
[39] Δ is required not to push or pop $\perp$ symbols; in this definition non-deterministic moves are allowed, but for our verification purpose, provided that the n-CPDS is obtained from a n-CPDA without output moves, it will be deterministic in the sense that we discussed in Section 3.2.

to *internal* moves: we have the transition $(q, s) \xrightarrow{(q',\theta)} (q', s')$ iff $s' = \theta(s)$ and $(q, s, q', \theta) \in \Delta$.

Definition 18 (n-CPDS configuration graph)**.** Given a n-CPDS, its configuration graph is the graph:

- whose vertices are the *reachable configurations*
- and whose (labelled) edges are the one induced by the transition relation $\xrightarrow{(q',\theta)}$ over reachable configuration.

Example 11 (A 2-CPDS and its transition graph)**.** We consider a 2-CPDS on the stack alphabet $\Sigma = \{\perp, a\}$ and with states in $Q = \{0, 1, 2)\}$. Its transition set contains the following rules: $(0, \perp, 1, push_2)$, $(0, a, 1, push_2)$, $(1, \perp, 2, push_1^{a,2})$, $(1, a, 2, push_1^{a,2})$, $(2, a, 1, push_2)$, $(2, a, 0, collapse)$. A part of its configuration graph follows:



$(0, [\,[\perp]\,])$
$push_2$
$(1, [\,[\perp]\,[\perp]\,])$
$collapse$
$push_1^{a,2}$
$push_2$
$(1, [\,[\perp]\,[\perp a]\,[\perp a]\,])$
$(2, [\,[\perp]\,[\perp a]\,])$
$push_1^{a,2}$
$(2, [\,[\perp]\,[\perp a]\,[\perp a\,a]\,])$
$push_2$
$\ldots$
$collapse$
$(0, [\,[\perp]\,[\perp a]\,])$
$push_2$
$\ldots$

Verification parity games over n-CPDS configurations graphs Parity games can easily be defined over n-CPDS configuration graphs: given a *partition* of states of Q beetween states belonging to Abélard and states belonging to Eloïse, there is an induced partition of vertices, where one vertex is owned by the owner of its control state. A *priority function* Ω over Q also induces a coloring of the vertices, leading to a parity game. Using results from Section 2.3 and Section 4.1, it is then equivalent to decide whether Eloïse has a winning strategy from the initial configuration of a n-CPDS parity game or to decide whether ϕ holds at the initial configuration of the n-CPDS transition graph. And, using the equi-expressivity of n-CPDA[40] and order-n recursion schemes, we obtain:

Theorem 6. *The problem of deciding whether Eloïse has a winning strategy from the initial configuration of a given n-CPDS parity game is equivalent to the problem of deciding whether a given modal μ-calculus formula ϕ holds at the root of a tree generated by an order-n recursion scheme.*

Sketch of a strategy for solving n-CPDS parity games. In Section 6 of [5], an interesting process of *order reduction* for the n-CPDS parity games is introduced, building from a given n-CPDS parity game $\mathcal{G}$ an equivalent $(n-1)$-CPDS parity game $\hat{\mathcal{G}}$, in the sense that Eloïse has a winning strategy from the root of $\mathcal{G}$ iff she has a winning strategy from the root of $\hat{\mathcal{G}}$, and that from a winning strategy for one of these two games one can effectively build one for the other game. We only give there some intuitions about this order reduction, details may be found in [5].
The idea is first to enrich $\mathcal{G}$'s configurations by building an equivalent n-CPDS parity game $\bar{\mathcal{G}}$ which is obtained by working on a bigger stack alphabet: instead of just having symbols from the stack alphabet of $\mathcal{G}$ in the n-stack of $\bar{\mathcal{G}}$, we have tuples stocking (among other things that mainly have a technical reason of being) information about the minimal colour encountered since the creation[41] of the top i-stack, for every $1 \leq i \leq n-1$. This extra information makes the new game $\bar{\mathcal{G}}$ *collapse-rank aware*: when the top_1 symbol links to a stack of order $n-1$, then just reading this top_1 symbol one can determinate the minimal colour encountered since the last time the n-CPDS was in a configuration $(q', collapse(s))$, *i.e.* since the last configuration where the n-stack was equal to what collapsing the current one would give.
To build a $(n-1)$-CPDS parity game $\hat{\mathcal{G}}$ equivalent to the original order-n one (that is $\bar{\mathcal{G}}$, the collapsible-rank aware system we just built), we have to consider only one order-$(n-1)$ stack from the n-stack of the original automaton. Thus we cannot access to the $(n-1)$-stacks situated below the one played in the equivalent order-$(n-1)$ game. The idea is then that a play over $\bar{\mathcal{G}}$ can be factorized in finite segments that does not access $(n-1)$-stacks below a given one. We first need to define *stack height* as the number of $(n-1)$-stacks in a n-stack. Then, given a play $\Lambda = s_0 s_1 \cdots$, we define a set $Steps_\Lambda$ which is the set of indices of positions of the play such that no configuration of smaller stack height than the one at the position

[40] n-CPDS are not expressive in the sense that they do not generate anything, but in the case of traversal-computing n-CPDAs, the computed branch may be reconstructed very easily by just outputing the symbols of the tree alphabet when they are pushed to the n-stack: thus in our verification problem the n-CPDS configurations can be used equivalently.

[41] An i-stack is considered to be created when obtained by a $push_{i+1}$ move; the stacks contained inside this i-stack keep the "creation time" they had before duplication.

given by the indice is reached later in the play. This induces a finite factorization of Λ beetwen two kind of finite sequences of consecutive moves: considering i and j two consecutive[42] indices of $Steps_\Lambda$, with $i < j$, we define:

- *bumps*, which are sequences of consecutive positions of Λ starting on v_i, ending on v_j, where the stack height is stricly higher beetween the ends,
- and *stairs*, where v_j reaches a stack height increased by one[43].

Each of these sequences of moves being finite, the parity condition over $\bar{\mathcal{G}}$ is then fulfilled iff the sequence of minimal colours in each sequence of the factorization satisfies it. The idea is then to play on a $(n-1)$-stack of the original game $\bar{\mathcal{G}}$, which is the one corresponding to the minimal stack height at the current position of the game. Moves inside this $(n-1)$-stack are perfomed, as well as *collapse* operations that point inside it (all these moves are called *trivial bumps*, because they are bumps of length 2); *stair* moves are also performed in the new game: the $(n-1)$-stack of $\bar{\mathcal{G}}$ we were playing on will not be used later in the play, so the new game simulates from now the $(n-1)$-stack that is just over it in the original n-stack. The problem is now to simulate properly non-trivial bumps, as we cannot create new $(n-1)$-stacks to play on in the new game. The idea is then that, instead of performing "for real" these non-trivial bumps, Eloïse gives Abélard information about how the simulation of the non-trivial bump ends, and which was the minimal encountered colour in the simulation of the bump, returning at the stack height that corresponds to the $(n-1)$-stack the new game has - the collapse-rank awareness then allows Eloïse to compute properly this information. If Abélard suspects Eloïse of missimulating the bump or returning a wrong minimal colour, he then has the possibility to ask for the real performing of the simulation; he wins if Eloïse tried to fool him and looses if he doubted of a real information. Else, Abélard can choose to trust Eloïse, and then the simulation continues considering the next sequence of moves in the factorization of the play.
This process leads to an exponentially bigger $(n-1)$-CPDS parity game $\tilde{\mathcal{G}}$ which is equivalent to the original n-CPDS parity game $\bar{\mathcal{G}}$. Iterating the transforms reduces our problem to the case of non-collapsible pushdown automata, which was studied by Walukiewicz in [10].

Complexity result. First of all, the transformation giving $\bar{\mathcal{G}}$ from $\mathcal{G}$ makes the game exponentially bigger. Then, every order reduction makes the equivalent game of lesser order exponentially bigger, and the case of the order-1 games is the

[42]In the sense that there is no $k \in Steps_\Lambda$ such that $i < k < j$.

[43]Note that it means that the automaton reached a new minimal stack height, in the sense that at no further position the play may go back to a lower stack height than this new one.

one of pushdown parity games[44] whose solving was proved EXPTIME-complete by Walukiewicz in [10]. So we have that solving a n-CPDS parity game is n-EXPTIME complete, and this gives another proof of Theorem 5, as well as another method for performing verification over order-n recursion schemes.

5 Conclusions and further directions

In this paper we have presented the two game-semantics based approaches for the modal μ-calculus model-checking over trees generated by recursion schemes. A new interesting approach by Ong and Kobayashi, developped in [7], uses *type systems* to prove the same result, without having to translate the original recursion scheme with the RHS transform: a type system is built, in which a recursion scheme is typable iff its generated tree satisfies a given modal μ-calculus formula. The resulting proof is comparatively easier to understand, and leads to some efficient - yet still in development - model-checking algorithm.

[44]Remember that order-1 links are just links to the previous stack symbol: so the *collapse* operation is just the pop_1, and thus collapsing adds no expressivity at order 1.

References

[1] S. Abramsky and G. McCusker. Game semantics. In H. Schwichtenberg and U. Berger, editors, *Computational Logic: Proceedings of the 1997 Marktoberdorf Summer School*, Lecture Notes in Computer Science, pages 1–56. Springer-Verlag, 1999.

[2] J.C. Bradfield and C.P. Stirling. Modal logics and mu-calculi: an introduction. In A. Ponse J. Bergstra and S. Smolka, editors, *Handbook of Process Algebra*, pages 293–330. Elsevier, 2001.

[3] E. Allen Emerson and Charanjit S. Jutla. Tree automata, mu-calculus and determinacy (extended abstract). In *FOCS*, pages 368–377. IEEE, 1991.

[4] Charles Grellois. Modal μ-calculus model-checking games over apt and n-cpds as a consequence of game semantics of higher-order recursion schemes (version with appendix). `http://student.grellois.fr/hors.pdf`.

[5] Matthew Hague, Andrzej S. Murawski, C.-H. Luke Ong, and Olivier Serre. Collapsible pushdown automata and recursion schemes. In *LICS*, pages 452–461. IEEE Computer Society, 2008.

[6] Marcin Jurdzinski. Small progress measures for solving parity games. In Horst Reichel and Sophie Tison, editors, *STACS*, volume 1770 of *Lecture Notes in Computer Science*, pages 290–301. Springer, 2000.

[7] Naoki Kobayashi and C.-H. Luke Ong. A type system equivalent to the modal mu-calculus model checking of higher-order recursion schemes. In *LICS*, pages 179–188. IEEE Computer Society, 2009.

[8] Dexter Kozen. Results on the propositional mu-calculus. *Theor. Comput. Sci.*, 27:333–354, 1983.

[9] C.-H. Luke Ong. On model-checking trees generated by higher-order recursion schemes. In *LICS*, pages 81–90. IEEE Computer Society, 2006.

[10] Igor Walukiewicz. Pushdown processes: Games and model-checking. *Inf. Comput.*, 164(2):234–263, 2001.

[11] Thomas Wilke. Alternating tree automata, parity games, and modal mu-calculus. *Bull. Belg. Math. Soc.*, 8(2):359–391, 2002.