# Application de la sémantique des jeux à la vérification des schémas de récursion d'ordre supérieur

**Charles Grellois**

*ENS Cachan*
`charles.grellois@ens-cachan.fr`

RÉSUMÉ. *Dans cette étude, nous étudions comment la sémantique des jeux a pu venir s'immiscer de façon surprenante dans la question de la complexité de la vérification des schémas de récursion d'ordre supérieur, et permettre pour la première fois d'en donner la complexité lorsque la logique utilisée est le $\mu$-calcul modal. Nous introduisons les bases de la sémantique des jeux, puis la traduction des schémas de récursion dans un cadre où elle s'avère utile pour comprendre la dynamique en jeu. Après des rappels sur le $\mu$-calcul modal et les automates alternants à parité, nous utilisons conjointement sémantique des jeux et techniques de vérification* via *des jeux de parité pour en déduire la complexité de ce problème de vérification.*

ABSTRACT. *In this survey, we have a look at how game semantics suprisingly helped determining the complexity of the model-checking of modal $\mu$-calculus formulæ over higher-order recursion schemes. We introduce elements of games semantics, then explain the traduction of recursion schemes into a framework where it can be used for a better understanding of the dynamics of the scheme. We then give a brief introduction to modal $\mu$-calculus and its link with alternating parity automata, and use these tools from verification together with game semantics to deduce parity games whose solving model-checks recursion schemes. The complexity of parity games resolution then gives the complexity of the model-checking task.*

MOTS-CLÉS : *Sémantique des jeux, vérification, schémas de récursion d'ordre supérieur, automates alternants, CPDA, jeux de parité*

KEYWORDS: *Game semantics, verification, higher-order recursion schemes, alternating automata, CPDA, parity games*

1ère soumission à *Technique et science informatiques*, le 7 décembre 2011

## Introduction

En 1981, Clarke, Emerson et Sifakis inventèrent la *vérification de modèle*. Cette nouvelle technique de vérification automatisée, consistant en une abstraction du programme à vérifier au moyen d'un modèle sur lequel on testera ensuite la véracité d'une formule logique par des méthodes formelles, a depuis pris un essor considérable. Un an plus tard, Damm (Damm, 1982) définissait les grammaires d'ordre supérieur, structures mathématiques permettant la génération de structures potentiellement infinies, dont les *schémas de récursion d'ordre supérieur* sont un cas particulier générant des structures arborescentes. A leur introduction, ceux-ci étaient voués à donner des sémantiques aux langages de programmation ; ils sont cependant revenus au premier plan au début des années 2000, mais cette fois-ci comme modèles pour la vérification. Leur intérêt fut renforcé par la découverte par Kobayashi en 2005 (Igarashi *et al.*, 2005) de la possibilité de réduire le problème de la vérification de l'utilisation des ressources par les programmes fonctionnels au problème de la vérification sur les modèles donnés par des schémas de récursion.
En 1969, Scott et De Bakker (Scott *et al.*, 1969) définirent le *$\mu$-calcul modal*, une logique dotée d'opérateurs de point fixe et de modalités. En l'absence de publication, celle-ci dut attendre le travail de Kozen en 1982 (Kozen, 1983) pour prendre son essor, et en 1991 Emerson et Jutla (Emerson *et al.*, 1991) firent le lien entre cette logique et un type particulier d'automates qui lui est équivalent, les *automates alternants à parité*, qui utilisent à la fois une structure d'automate d'arbres et une structure de jeux de parité.
Le problème de la complexité de la vérification des programmes modélisés par des schémas de récursion d'ordre supérieurs et sur lesquels on exprime les formules devant être satisfaites à l'aide du $\mu$-calcul modal est resté ouvert jusqu'en 2006, quand Ong (Ong, 2006) apporta la solution à l'aide d'outils théoriques apparemment décorrélés puisqu'issus de la *sémantique des jeux*. Cette théorie, cherchant à donner aux programmes une sémantique à l'aide d'un modèle de jeu interactif, a des origines diverses, aussi bien du côté de la philosophie où elle émergea dans les années 1950 des travaux parallèles de Lorenzen et de Hintikka, que du côté de l'informatique où les premières traces remontent à 1977 avec l'introduction par Joyal de la catégorie des jeux de Conway (Joyal, 1977) ; l'introduction de la logique linéaire en 1987 par Girard (Girard, 1987), ses travaux ultérieurs sur la géométrie de l'interaction et sur la ludique, ainsi que ceux de Blass (Blass, 1992) motivèrent le développement de cette approche interactive de la logique, et de son penchant par la correspondance de Curry-Howard, le calcul. Deux modèles de jeux apparurent à peu près en même temps pour donner une sémantique au langage abstrait PCF, composé d'un $\lambda$-calcul avec récursion et opérations arithmétiques : Abramsky, Malacaria et Jagadeesan introduisirent les jeux dits AJM en 1994 (Abramsky *et al.*, 1994), tandis qu'Hyland et Ong développaient les jeux HO (Hyland *et al.*, 2000). Le développement de ces modèles de jeux permit notamment de donner des sémantiques à des langages abstraits (PCF avec contrôle et effets de bord, Idealized Algol), mais également à des fragments de la logique linéaire.
Cette étude vise principalement à introduire à la contribution de Ong (Ong, 2006)

réunissant vérification et sémantique des jeux. On étudiera également une approche alternative et cependant similaire, développée dans (Hague *et al.*, 2008). Pour cela, nous commencerons par introduire en Section 1 les bases de la sémantique des jeux, utiles (sans être cependant absolument indispensables) au lecteur pour comprendre les motivations et la profondeur de l'approche développée par la suite. En Section 2, nous introduirons les modèles des programmes que nous voulons vérifier, qui sont les schémas de récursion d'ordre supérieur, puis leur traduction dans un $\lambda$-calcul, appelée transformation RHS, qui permettra ensuite l'interprétation de leur dynamique à l'aide de la sémantique des jeux. En Section 3, nous introduirons le $\mu$-calcul modal, logique adaptée aux productions des schémas de récursion qui permettra de décrire les comportements à vérifier sur les modèles de programmes, puis les automates alternants à parité et leur lien avec la vérification du $\mu$-calcul modal. En Section 3, nous verrons deux moyens différents, correspondant au cœur des approches développées dans (Ong, 2006) et (Hague *et al.*, 2008), de simuler le comportement des schémas de récursion d'ordre supérieur à l'aide de leur traduction en sémantique des jeux, qui s'avèrera elle d'ordre 0 et génèrera donc des arbres réguliers. En Section 4, nous verrons comment ces deux approches mènent à des jeux de vérification à parité, et permettent toutes deux de déduire la complexité du problème de vérification de problèmes modélisés par des schémas de récursion d'ordre supérieur et dont les propriétés à vérifier sont exprimées en $\mu$-calcul modal.
On supposera une certaine familiarité du lecteur avec le $\lambda$-calcul : au besoin, on pourra consulter par exemple le cours de Jean Goubault-Larrecq donné à l'ENS de Cachan (Goubault-Larrecq, 2011).

Sur la paternité de ce travail.

Ceci est une *étude* : la contribution de l'auteur se réduit à une compilation et à une mise en forme de résultats, ainsi qu'à l'introduction de nouveaux exemples ou à l'étoffement d'exemples existants. Les articles majoritairement utilisés pour la rédaction de cette étude sont (Ong, 2006), (Hague *et al.*, 2008), (Abramsky *et al.*, 1999) et (Wilke, 2002).

Remerciements.

Je voudrais remercier Luke Ong pour son encadrement à Oxford durant l'été 2010, son groupe pour la qualité des discussions que nous avons eues, ainsi que Paul-André Melliès, qui me fit découvrir le domaine. Je suis très reconnaissant à William Blum d'avoir développé le logiciel *HOG*[1], très pratique pour la traduction des schémas de récursion dans un langage de sémantique des jeux et leur exploitation.

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

## 1. Une brève introduction à la sémantique des jeux

La sémantique des jeux offre une interprétation dynamique des langages de programmation : au lieu d'un modèle statique, tel qu'une catégorie dotée de certaines propriétés, on s'intéresse à des jeux coopératifs où l'interaction a lieu entre un programme, représenté par la lettre $P$, et son environnement, noté $E$. L'environnement ouvre le jeu en demandant au programme sa valeur. Si le programme s'avère trivial, sa réponse sera immédiate : il suffira de répondre à l'environnement par la valeur de retour. Dans des cas plus complexes, il faudra demander à l'environnement quelles sont les valeurs des variables libres : le programme répondra donc non pas par une valeur de retour, mais par une nouvelle question, interprétée comme la demande de la valeur d'un argument. Celle-ci pourrait demander à son tour un calcul : en ce cas, l'environnement est en charge de l'effectuer, et joue alors localement comme un programme : il y a échange des rôles. Voyons les choses plus formellement : tout d'abord, le support du jeu est une arène.

**Définition 1 (Arène)** *Une arène est une structure* $< M_A, \lambda_A, \vdash_A >$ *telle que :*

– $M_A$ *est un ensemble de* coups*,*

– $\lambda_A : M_A \to \{E, P\} \times \{Q, R\}$ *est une* fonction d'étiquetage *: elle indique si chaque coup potentiel est voué à être joué par l'environnement* $E$ *ou le programme* $P$*, et s'il s'agit pour lui de poser une nouvelle question (*$Q$*) ou de répondre à une question (*$R$*).*

– $\vdash_A$ *est une relation entre* $M_A \cup \{\bullet\}$ *(*$\bullet$ *désignant l'absence de coup) et* $M_A$*, appelée* relation d'activation*, et telle que :*

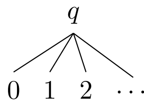
*- Si* $\bullet \vdash m$*, alors* $\lambda_A(m) = EQ$*, et seul* $\bullet$ *peut activer* $m$ *(le coup initial est joué par l'environnement, c'est une question ; quant à la seconde condition, elle provient du fait que* $\vdash_A$ *donne à* $M_A$ *une structure de forêt, dont les racines correspondent aux coups initiaux)*

*- Si* $m \vdash_A n$ *et que* $n$ *correspond à une réponse (c'est à dire si la seconde composante de* $\lambda_A(n)$ *est* $R$*), alors* $m$ *est une question - l'idée étant que dans la structure de l'arène les réponses ne viennent qu'après des questions*

*- Si* $m \neq \bullet$ *et que* $m \vdash_A n$*, alors* $m$ *et* $n$ *sont voués à être joués par des joueurs différents : la structure d'une arène est alternée entre les joueurs à mesure que l'on s'éloigne de la racine des arbres qui la composent.*

Avant de continuer, notons $\bar{\lambda_A}$ la fonction qui inverse les propriétaires de chaque coup sans en changer la nature. Prenons maintenant un exemple simple d'arène :

**Exemple 1 (L'arène des entiers)** *L'arène* $\mathbb{N}$ *est telle que suit :*



Cette représentation signifie que l'arène $\mathbb{N}$ a pour unique coup initial $q$ (c'est la *question* initiale de l'environnement, demandant au programme sa valeur, c'est à dire l'entier qu'il représente), et que $q$ active tous les autres sommets, permettant au programme de répondre et de donner la valeur entière à laquelle il correspond.

Contrairement à ce que semble suggérer cet exemple, on s'apercevra bientôt que les arêtes données par la relation d'activation de l'arène ne fournissent pas le graphe sous-jascent au jeu ; celui-ci n'est d'ordinaire pas explicité, les parties étant définies à l'aide de la relation d'activation. Elles correspondent à ce que l'on appelle des *suites justifiées licites*.

**Définition 2 (Suite justifiée)** *Une suite justifiée sur une arène* $< M_A, \lambda_A, \vdash_A >$ *est une suite de coups de* $A$*, commençant par un coup initial, et dont tout coup non initial* $m$ *est relié par un pointeur, dit* pointeur de justification*, à un coup* $n$ *le précédant dans la suite et vérifiant* $n \vdash_A m$*. On dit que* $n$ *justifie* $m$*.*

**Exemple 2** *Sur l'arène* $\mathbb{N}$*,* $q\ 0$ *est une suite justifiée. Intuitivement, elle représente le programme codant la constante* $0$*.*

On ne considère qu'un sous-ensemble de ces suites, l'idée étant qu'une partie doit voir les joueurs alterner leurs coups, mais également qu'ils doivent jouer selon leur *vue*, c'est à dire qu'ils ne doivent tenir compte que de la réponse à leurs questions, pas de la façon dont elle a été obtenue.

**Définition 3 (Vue)** *Etant donnée une suite justifiée* $s$*, on définit la vue* $\lceil s \rceil$ *du programme et celle* $\lfloor s \rfloor$ *de l'environnement comme suit :*

- $\lceil \epsilon \rceil \ = \ \epsilon$
- $\lceil s \cdot m \rceil \ = \ \lceil s \rceil m$ *si* $m$ *appartient à* $P$
- $\lceil s \cdot m \rceil \ = \ m$ *si* $\bullet \vdash m$
- $\lceil s \cdot m \cdot t \cdot n \rceil \ = \ \lceil s \rceil m \cdot n$ *si* $n$ *appartient à* $E$
- $\lfloor \epsilon \rfloor \ = \ \epsilon$
- $\lfloor s \cdot m \rfloor \ = \ \lfloor s \rfloor m$ *si* $m$ *appartient à* $E$
- $\lfloor s \cdot m \cdot t \cdot n \rfloor \ = \ \lfloor s \rfloor m \cdot n$ *si* $n$ *appartient à* $P$

Les définitions sont duales ; il semble y avoir une règle de plus pour le programme, mais sa duale est en fait contenue dans la deuxième règle pour l'environnement. La notion de vue nous permet alors de définir celle de suite justifiée licite.

**Définition 4 (Suite justifiée licite)** *Une suite justifiée* $s$ *est dite* licite *lorsqu'elle satisfait les conditions additionnelles suivantes :*

*– Les joueurs alternent : pour toute factorisation $s = s_1 \cdot m \cdot n \cdot s_2$ où $m$ et $n$ sont deux coups et $s_1$ et $s_2$ deux suites de coups, $m$ et $n$ n'appartiennent pas au même protagoniste.*

*– Si $tm$ est un préfixe de $s$, avec $m$ un coup appartenant au programme, alors $m$ est justifié par un sommet apparaissant dans $\lceil t \rceil$.*

*– Si $tm$ est un préfixe de $s$, avec $m$ un coup non-initial appartenant à l'environnement, alors $m$ est justifié par un sommet apparaissant dans $\lfloor t \rfloor$.*

L'ensemble des suites justifiées licites d'une arène $A$ est noté $L_A$. Dans une suite justifiée, on dit qu'un coup $n$ est *héréditairement justifié* par $m$ si $m \vdash^* n$ (où l'étoile représente l'itération). Si $s$ est une telle suite, on note $s \upharpoonright n$ la sous-suite de $s$ ne contenant que les coups héréditairement justifiés par $n$, et plus généralement $s \upharpoonright S$ la sous-suite de $s$ ne contenant que les coups héréditairement justifiés par un coup de l'ensemble $S$. On peut alors définir un *jeu* :

**Définition 5 (Jeu)** *Un jeu $A$ est la donnée d'une structure $< M_A, \lambda_A, \vdash_A, P_A >$ telle que :*

*– $< M_A, \lambda_A, \vdash_A>$ est une arène,*

*– $P_A$ est un sous-ensemble non vide de $L_A$, clos par préfixe, et tel que si $s \in P_A$ et que $I$ est un ensemble de mouvements initiaux, $s \upharpoonright I \in P_A$. On appelle $P_A$ l'ensemble des* positions autorisées *de $A$.*

On peut maintenant définir la notion de *stratégie* sur un tel jeu.

**Définition 6 (Stratégie)** *Une stratégie $\sigma$ pour un jeu $A$ est un ensemble non vide de positions autorisées de $A$, toutes de longueur paire, et satisfaisant les conditions suivantes :*

*– $sab \in \sigma \Rightarrow s \in \sigma$*

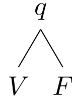
*– $(sab, sac) \in \sigma^2 \Rightarrow b = c$ et $b$ et $c$ sont justifiés par le même coup[2].*

L'idée de la sémantique des jeux est de représenter un programme d'un certain type comme une stratégie sur une arène correspondant à ce type. Le comportement du programme n'est alors pas donné par une fonction, comme notamment en théorie des domaines, mais par l'ensemble de ses *traces d'exécution* ; par exemple, celle du programme constant 0 sur l'arène $\mathbb{N}$ a pour trace :

$$\begin{array}{cc} \mathbb{N} & \\ q & E \\ 0 & P \end{array}$$

2. Autrement dit, les stratégies que l'on considère ici sont *déterministes*. Il existe des versions non-déterministes de la sémantique des jeux, voir par exemple (Harmer *et al.*, 1999).

La stratégie correspondant à ce programme est en effet très simple, elle consiste simplement à toujours répondre $0$ à la question de l'environnement, il n'y a qu'un seul comportement attendu - ce qui ne serait pas nécessairement le cas si, par exemple, le programme avait à prendre un argument. Pour construire les jeux correspondants aux types applicatifs, on peut partir de la structure des arènes des types de base - comme $\mathbb{N}$, que nous avons déjà vue, ou alors $\mathbb{B}$ l'arène des booléens :




pour construire le tenseur, l'application, et l'exponentielle ; il suffit ensuite de donner un ensemble de positions autorisées cohérent à la nouvelle arène pour obtenir de nouveaux jeux.

**Définition 7 (Tenseur de jeux)** *Etant donnés deux jeux $A$ et $B$, on définit leur tenseur comme suit :*

$$\begin{array}{rcl} M_{A\otimes B} & = & M_A + M_B \\ \lambda_{A\otimes B} & = & [\lambda_A, \lambda_B] \\ \bullet \vdash_{A\otimes B} n & \Leftrightarrow & \bullet \vdash_A n \textit{ ou } \bullet \vdash_B n \\ m \vdash_{A\otimes B} n & \Leftrightarrow & m \vdash_A n \textit{ ou } m \vdash_B n \\ P_{A\otimes B} & = & \{s \in L_{A\otimes B} | s \upharpoonright A \in P_A \textit{ et } s \upharpoonright A \in P_A\} \end{array}$$

L'idée du tenseur de deux jeux est simple : on met les deux arènes côte à côte pour en former une nouvelle, voici par exemple $\mathbb{N} \otimes \mathbb{N}$ :

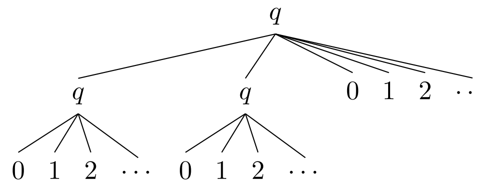
et sur cette nouvelle arène les positions autorisées sont celles dont la restriction à l'arène de gauche sont autorisées dans l'arène de gauche, et de même pour l'arène de droite. En fait, le tenseur de deux jeux correspond juste à la possibilité de jouer *en parallèle* sur ces deux jeux.

**Définition 8 (Jeux pour les types application)**
*Etant donnés deux jeux $A$ et $B$, on définit $A \rightarrow B$ comme suit :*

$$\begin{array}{rcl} M_{A\rightarrow B} & = & M_A + M_B \\ \lambda_{A\rightarrow B} & = & [\overline{\lambda}_A, \lambda_B] \\ \bullet \vdash_{A\rightarrow B} n & \Leftrightarrow & \bullet \vdash_B n \\ m \vdash_{A\rightarrow B} n & \Leftrightarrow & m \vdash_A n \textit{ ou } m \vdash_B n \textit{ ou } m \neq \bullet \textit{ et } \bullet \vdash_B m \textit{ et } \bullet \vdash_A n \\ P_{A\rightarrow B} & = & \{s \in L_{A\rightarrow B} | s \upharpoonright A \in P_A \textit{ et } s \upharpoonright A \in P_A\} \end{array}$$

L'idée d'une telle application d'un jeu dans un autre est que l'on commence par jouer dans le jeu $B$, qui correspond au type de la valeur de retour d'une interaction réussie ; la partie s'ouvre lorsque l'environnement y pose une question - lorsqu'il demande au programme sa valeur. Ensuite, le programme peut au besoin visiter l'arène $A$, qui est celle de son argument, pour l'obtenir : il peut ainsi y poser la question initiale (on notera que l'utilisation de la fonction d'étiquetage $\overline{\lambda}_A$ renverse la polarité des coups dans l'arène $A$ correspondant à l'entrée du programme : en effet, les paramètres sont du ressort de l'environnement, s'ils nécessitent un calcul, c'est donc à lui de le mener), à laquelle l'environnement devra répondre (éventuellement après calcul) par la valeur de l'argument (de type $A$) du programme. Le programme pourra alors utiliser cet argument pour rendre sa valeur.

**Exemple 3 (L'arène $\mathbb{N} \otimes \mathbb{N} \to \mathbb{N}$)**




On remarquera que la polarité des coups (c'est à dire leur propriétaire) alternant dans l'arène, tous les coups de chaque niveau représenté dans cet arbre appartiennent à un même joueur - on voit bien sur cet exemple l'inversion de polarité des coups dûe à l'application : il faut avoir posé la question initiale dans $\mathbb{N}$ pour avoir le droit d'éventuellement[3] jouer dans $\mathbb{N} \otimes \mathbb{N}$. Considérons une stratégie sur ce jeu - c'est à dire un programme de type $\mathbb{N} \otimes \mathbb{N} \to \mathbb{N}$ : l'addition. Ses traces[4] sont, pour $(m, n) \in \mathbb{N}^2$ :

| $\mathbb{N}$ | $\otimes$ | $\mathbb{N}$ | $\to$ | $\mathbb{N}$ | |
|---|---|---|---|---|---|
| | | | | $q$ | $E$ |
| $q$ | | | | | $P$ |
| $n$ | | | | | $E$ |
| | | $q$ | | | $P$ |
| | | $m$ | | | $E$ |
| | | | | $n + m$ | $P$ |

Etant donnés deux jeux $A \to B$ et $B \to C$, on peut définir $A \to B \to C$ sans encombre. Il est cependant intéressant de remarquer qu'il est possible de composer

3. Un programme n'est pas obligé d'utiliser ses arguments : il pourrait directement choisir une réponse entière et terminer.
4. Pour être plus précis, il s'agit ici de l'addition dans une évaluation commençant par le premier argument, puis prenant le second. Une autre stratégie, correspondant à collecter les arguments dans l'ordre inverse, donne un programme différent mais calculant la même chose.

deux stratégies $\sigma : A \to B$ et $\tau : B \to C$ en une stratégie $\sigma;\tau : A \to C$. Sur les traces, l'idée est simple : mettons par exemple que l'on veuille effectuer l'addition de deux nombres entiers $n$ et $m$ puis en prendre le successeur. Ceci a pour trace :

$$\begin{array}{ccccccccc}
\mathbb{N} & \otimes & \mathbb{N} & \to & \mathbb{N} & \to & \mathbb{N} & \\
 & & & & & & q & E \\
 & & & & q & & & P \\
q & & & & & & & E \\
n & & & & & & & P \\
 & & q & & & & & E \\
 & & m & & & & & P \\
 & & & & n+m & & & E \\
 & & & & & & n+m+1 & P
\end{array}$$

Pour obtenir les traces correspondant à la stratégie composée, il suffit d'oublier l'interaction qui a eu lieu dans le jeu du milieu : ceci donne les traces :

$$\begin{array}{cccccc}
\mathbb{N} & \otimes & \mathbb{N} & \to & \mathbb{N} & \\
 & & & & q & E \\
q & & & & & P \\
n & & & & & E \\
 & & q & & & P \\
 & & m & & & E \\
 & & & & n+m+1 & P
\end{array}$$

Plus formellement, prenons une suite de coups $u$ sur les jeux $A$, $B$ et $C$, dont tous les coups sont justifiés mis à part ceux initiaux dans $C$. On dit que $u$ est une *bonne interaction* pour les jeux $A$, $B$ et $C$ si sa projection sur les jeux $B$ et $C$[5] est une position autorisée de $B \to C$, et de même pour la projection sur $A$ et $B$.

**Définition 9 (Composition de stratégies)** *Etant données $\sigma : A \to B$ et $\tau : B \to C$, on définit leur composée $\sigma;\tau : A \to C$ comme l'ensemble des projections sur $A$ et $C$ des bonnes interactions pour les jeux $A$, $B$ et $C$ dont les projections sur $A$ et $B$ sont dans $\sigma$, et celles sur $B$ et $C$ dans $\tau$.*

Ceci exprime exactement ce que l'on a vu sur les traces : une stratégie composée $\sigma;\tau$ est une stratégie qui joue $\sigma$ “à gauche”, $\tau$ “à droite”, et oublie ensuite le “milieu” par projection sur $A$ et $C$. Il est intéressant de remarquer que la composition de stratégies permet de réaliser l'application de termes : mettons que l'on veuille appliquer la fonction successeur $\lambda x.x+1 : \mathbb{N} \to \mathbb{N}$ à un entier $n \in \mathbb{N}$. On peut voir $n$ comme une stratégie sur l'arène $\mathbb{N} \cong \bot \to \mathbb{N}$ (où $\bot$ est l'arène vide), on a donc deux stratégies donnant les traces suivantes :

5. C'est à dire, la sous-suite de $u$ dans laquelle les coups de $A$ ont été enlevés, ainsi que tout éventuelle justification d'un coup de $B$ ou $C$ par un coup de $A$.

$$\begin{array}{ccccccccc} & & & & \mathbb{N} & \rightarrow & \mathbb{N} & \\ \bot & \rightarrow & \mathbb{N} & & & & q & E \\ & & q & E & q & & & P \\ & & m & P & m & & & E \\ & & & & & & m+1 & P \end{array}$$

et ceci se compose en une stratégie sur $\bot \rightarrow \mathbb{N} \cong \mathbb{N}$, c'est à dire en un entier, qui correspond au successeur de $n$. On aurait pu, de même, utiliser la composition de stratégies pour réaliser l'application de termes d'ordre supérieur. La composition de stratégies admet une identité $A \rightarrow A$ : il s'agit de la stratégie du *perroquet*, consistant pour le programme à répéter exactement ce que lui dit l'environnement. Sur l'arène $A$ de droite, l'environnement demande $(q)$ la valeur du programme identité, le programme répète le $q$ de l'autre côté de la flèche ; l'environnement donne alors la valeur de l'argument, et le programme répond par cette valeur de l'autre côté de la flèche, donc là où on a ouvert la partie :

$$\begin{array}{cccc} A & \rightarrow & A & \\ & & q & E \\ q & & & P \\ m & & & E \\ & & m & P \end{array}$$

Une dernière construction utile sur les jeux est l'*exponentielle*. Intuitivement, elle permet de rejouer plusieurs fois sur une arène.

**Définition 10 (Exponentielle d'un jeu)** *Etant donné un jeu $A$, son* exponentielle*, notée $!A$, est le jeu de même arène sous-jascente, et dont l'ensemble des positions autorisées est :*

$$P_{!A} = \{s \in L_{!A} | \forall m, \bullet \vdash m \Rightarrow s \upharpoonright m \in P_A\}$$

On peut également définir un produit (correspondant au produit cartésien dans une catégorie de jeux et de leurs stratégies[6]) $\&$, dont l'effet est d'autoriser le jeu sur deux arènes, mais en réalité uniquement sur une seule : l'environnement choisit par son premier coup sur quelle arène aura lieu le jeu, et on ne peut plus ensuite disposer de l'autre arène. Le lecteur connaissant la logique linéraire voit probablement se profiler une analogie forte avec la sémantique des jeux, d'autant plus forte si l'on pose $A \multimap B = A \rightarrow B$, $A \Rightarrow B = !A \multimap B$ et $A^\bot = A \multimap \bot$.

6. Il y a en fait plusieurs telles catégories, selon la classe de stratégies considérée, et le type de flèche (linéaire ou intuitionniste), voir (Abramsky *et al.*, 1999, Section 3).

REMARQUE. — Sémantique des jeux et logique linéaire
L'essor de la sémantique des jeux commence réellement dans les années 1990, quand Blass (Blass, 1992) introduit un modèle de la logique linéaire à l'aide de la sémantique des jeux, et montre ainsi qu'on peut interpréter comme des jeux séquentiels les formules de la logique linéaire. Le problème de cette approche est que le modèle qu'elle fournit n'est pas une catégorie : la composition des stratégies n'y est pas associative. De fait, seuls des fragments de la logique linéaire sont à ce jour modélisés par la sémantique des jeux ; on en trouvera une description dans l'introduction de (Melliès, 2005).

REMARQUE. — Traces et suites justifiées
Les traces correspondent à des suites justifiées - celles de la stratégie correspondant au programme que l'on fait interagir avec l'environnement - dont on a enlevé les pointeurs de justification. On peut ainsi questionner l'utilité de ces pointeurs : nous n'en avons en effet pas eu besoin dans les exemples. Ceci vient du fait que la nécessité de tels pointeurs n'apparaît qu'à l'ordre[7] 3 ; considérons en effet les termes de Kierstead :

$$\lambda f.f(\lambda x.f(\lambda y.y)) \text{ et } \lambda f.f(\lambda x.f(\lambda y.x))$$

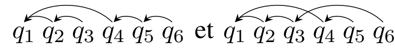
Ces deux termes sont de type $((\mathbb{N} \to \mathbb{N}) \to \mathbb{N}) \to \mathbb{N}$ (notons donc que $f$ est de type $(\mathbb{N} \to \mathbb{N}) \to \mathbb{N}$), et correspondent donc, en tant que programmes, à des stratégies sur l'arène correspondante. Leurs traces d'exécution commencent toutes deux ainsi (on a ajouté des indices aux questions pour clarifier l'explication à venir) :

$$\begin{array}{cccccccc}
((\mathbb{N} & \Rightarrow & \mathbb{N}) & \Rightarrow & \mathbb{N}) & \Rightarrow & \mathbb{N} & \\
 & & & & & & q_1 & E \\
 & & & & q_2 & & & P \\
 & & q_3 & & & & & E \\
 & & & & q_4 & & & P \\
 & & q_5 & & & & & E \\
q_6 & & & & & & & E
\end{array}$$

En effet, le jeu s'ouvre sur une question $q_1$ de l'environnement, sommant le programme de donner sa valeur. Celui-ci demande ($q_2$) pour cela à l'environnement la valeur de $f$, $E$ demande ($q_3$) à son tour à $P$ la valeur de l'argument de $f$, or ceci demande un nouveau calcul de $f$ : c'est à $P$ de l'effectuer, et il le commence en demandant ($q_4$) à l'environnement la valeur de l'argument de $f$, ce qui demande de calculer la fonction qui est en paramètre de $f$ ($\lambda y.y$ dans le premier cas et $\lambda y.x$ dans l'autre), qui commence par $q_5$ ("que vaut cette fonction"), qui amène naturellement à la question $q_6$ ("quel est la valeur à substituer à $y$ ?"). Le problème apparaît ici : la récursion ayant amené à jouer deux fois sur $(\mathbb{N} \Rightarrow \mathbb{N}) \Rightarrow \mathbb{N}$, on ne sait plus si la

7. Voir la définition 12.

réponse à $q_6$ doit être $x$ ou $y$. Avec des pointeurs de justification, cette trace commune à deux termes non équivalents donne lieu à deux suites justifiées différentes :

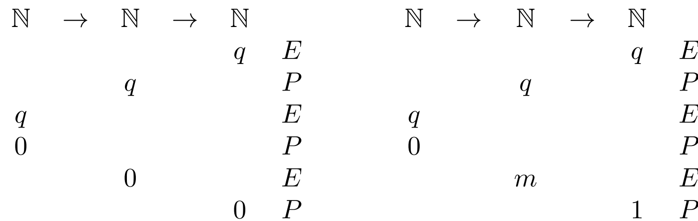



et l'information supplémentaire permet de retrouver de quelle variable on parle : de celle qui correspond à la première ou à la seconde visite de la sous-arène $(\mathbb{N} \Rightarrow \mathbb{N}) \Rightarrow \mathbb{N}$.

Nous avons maintenant de quoi créer, à partir de jeux représentant des types de base, une grande variété de types plus élaborés, sur lesquels joueront nos programmes - qui sont les stratégies sur ces types. Pour terminer cette introduction rapide à la sémantique des jeux, nous allons étudier deux types particuliers de stratégies : les *stratégies innocentes*, et les *stratégies bien parenthésées*.

**Définition 11 (Stratégies innocentes)** *Une stratégie $\sigma$ est* innocente *lorsqu'elle ne dépend que de sa vue : si $smn \in \sigma$, $t \in \sigma$, $tm \in P_A$ et $\lceil sm \rceil = \lceil tm \rceil$, alors $tmn \in \sigma$ (où l'on fait pointer $n$ dans $tmn$ vers le même coup que $n$ dans $smn$).*

Une stratégie innocente est une stratégie qui ne se préoccupe pas des calculs cachés par la vue, c'est à dire des calculs intermédiaires achevés. Si on se place par exemple sur l'arène $(\mathbb{N} \to \mathbb{N}) \to \mathbb{N}$ et qu'on considère la stratégie correspondant au programme prenant en entrée un terme de type $\mathbb{N} \to \mathbb{N}$ et rendant $0$ s'il s'évalue $0$ sur la valeur $0$ et $1$ sinon, a-t-on une stratégie innocente ? Cela semble le cas au vu des traces suivantes (avec $m > 0$) :



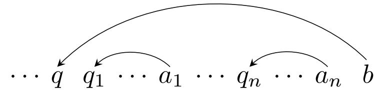
| $\mathbb{N}$ | $\to$ | $\mathbb{N}$ | $\to$ | $\mathbb{N}$ | |
|---|---|---|---|---|---|
| | | | | $q$ | $E$ |
| | | $q$ | | | $P$ |
| $q$ | | | | | $E$ |
| $0$ | | | | | $P$ |
| | | $0$ | | | $E$ |
| | | | | $0$ | $P$ |

| $\mathbb{N}$ | $\to$ | $\mathbb{N}$ | $\to$ | $\mathbb{N}$ | |
|---|---|---|---|---|---|
| | | | | $q$ | $E$ |
| | | $q$ | | | $P$ |
| $q$ | | | | | $E$ |
| $0$ | | | | | $P$ |
| | | $m$ | | | $E$ |
| | | | | $1$ | $P$ |

correspondant aux suites justifiées $q\ q\ q\ 0\ 0\ 0$ et $q\ q\ q\ 0\ m\ 1$. La vue du programme lorsqu'on lui rend $0$ ou $m$ est en effet $q\ q\ 0$ ou $q\ q\ m$, et sa réponse ne dépend que des informations contenues dans cette vue. Il faut cependant faire attention au fait que dans le cadre d'une stratégie innocente, on ne peut pas savoir si le terme de type $\mathbb{N} \to \mathbb{N}$ donné par l'environnement a effectivement utilisé son argument. Ici, cela ne change rien, si le terme n'utilise pas son argument c'est qu'il est constant, et notre programme répond bien ce qu'il doit répondre, et est effectivement innocent. Par contre, si l'on veut obtenir un programme déterminant si

un tel terme utilise son argument, on ne peut plus avoir d'innocence pour la stratégie correspondante : elle doit regarder des calculs cachés par la vue. L'idée est que les programmes correspondant aux stratégies innocentes sont les programmes *sans effets de bord* : si on ne le retient nulle part, on ne peut pas savoir si le calcul d'un terme a utilisé son argument[8].

Il existe une autre classe de stratégies intéressantes, les stratégies *bien parenthésées* : ce sont les stratégies dans lesquelles toute réponse du programme (mais ceci n'est pas exigé de l'environnement) est une réponse à la plus récente des questions encore ouvertes posées par l'environnement. Pour une stratégie bien parenthésée, les suites justifiées finissant par une réponse de $P$ sont donc de la forme suivante :




où les $a_i$ sont des réponses. L'idée est qu'une stratégie bien parenthésée correspond à un programme n'utilisant pas d'opérateurs de contrôle, c'est-à-dire, ne pouvant pas s'échapper d'une évaluation de fonction - au moyen d'une instruction comme `catch` ou `callcc` par exemple. Il existe des prototypes de langages issus du $\lambda$-calcul correspondant aux différentes classes de stratégies : quand il y a innocence et bon parenthésage, le langage est PCF ; on peut lui ajouter des références (perte de l'innocence) ou du contrôle (perte du parenthésage), voire les deux. On trouvera les définitions dans (Abramsky *et al.*, 1999, Section 4).

REMARQUE. — Jeux et sémantique des jeux
L'apparition du mot *jeu* dans *sémantique des jeux* peut porter à confusion : contrairement à ce qui se passe usuellement en théorie des jeux, il n'est pas ici question de joueurs égoïstes, rusés, ou de cadre compétitif. L'interaction décrite par le jeu fait intervenir deux *partenaires* plutôt que deux adversaires, même si leurs polarités les rendent duaux. En effet, ces protagonistes sont supposés parfaits et collaborent sans malice en vue de l'évaluation du programme dans un environnement donné : c'est d'ailleurs pour cela que la stratégie exhibée ci-dessus est innocente, s'il fallait douter de l'environnement et vérifier qu'il évalue bien le terme en entrée sur la valeur $0$, on ne pourrait plus se contenter de la vue pour répondre. L'inversion des polarités des coups dans l'arène source d'une application va également en faveur de cette conception de partenariat, l'échange des rôles supposant une confiance mutuelle absolue entre les deux joueurs[9].

8. L'astuce étant de passer au terme un argument correspondant en fait à un bout de code effectuant un effet de bord avant de ne passer pour valeur au terme un entier - voir (Abramsky *et al.*, 1999, Section 2.12).
9. On remarquera l'aspect très taoïste de cette dualité ...

## 2. Les schémas de récursion comme modèles de programmes

### 2.1. *Schémas de récursion d'ordre supérieur*

Afin de vérifier un programme, il est courant d'en considérer un *modèle* dans un certain formalisme, puis de vérifier que celui-ci satisfait une propriété logique donnée. Après avoir introduit dans cette section la variété de modèles que nous considèrerons dans la suite, les *schémas de récursion d'ordre supérieur*, nous définirons dans la suivante la logique dans laquelle exprimer les propriétés à vérifier. Les schémas de récursion d'ordre supérieur[10] permettent de décrire les programmes générant des arbres (potentiellement infinis) avec l'expressivité d'un $\lambda$-calcul simplement typé muni d'un opérateur de récursion et d'un ensemble de symboles de constantes non interprétés [11]. La définition de ces schémas fait appel à un système de types applicatifs simple : à partir d'un type de base $o$, on considère tous les types obtenus à l'aide de la flèche et de parenthèses, comme par exemple $(o \to o) \to o \to o$. Une notion importante sur ce système de types est celle d'*ordre*, qui caractérise la quantité maximale de $\beta$-réductions imbriquées pouvant survenir lors de la réduction d'un terme d'un type donné :

**Définition 12 (Arité et ordre d'un type)** *Tout type d'un système de types applicatifs simple admet une décomposition $A_1 \to \cdots \to A_n \to o$ ; on appelle $n$ l'*arité *de ce type et on définit son* ordre *comme suit :*

– $ordre(o) \;=\; 0$

– $ordre(A_1 \to \cdots \to A_n \to o) \;=\; 1 + \max(ordre(A_1), \cdots, ordre(A_n))$

Nous pouvons dorénavant définir les schémas, qui se comportent comme des grammaires à paramètres générant des termes, ou de façon équivalente des arbres.

**Définition 13 (Schéma de récursion d'ordre supérieur)** *Un schéma de récursion d'ordre supérieur est un quadruplet $G = < \Sigma, \mathcal{N}, \mathcal{R}, S >$, où :*

– $\Sigma$ *est un alphabet de* terminaux *munis d'une arité*

– $\mathcal{N}$ *est un ensemble fini de* non-terminaux[12]

– $S \in \mathcal{N}$ *est le* symbole initial*, de type de base $o$*

– $\mathcal{R}$ *est un ensemble fini de* règles de réécriture*, qui en contient une pour chaque non-terminal $F \;:\; A_1 \to \cdots \to A_n \to o$, de la forme $F\xi_1 \cdots \xi_n \to e$ où $\xi_i \;:\; A_i$ et $e : o$.*

*On définit l'*ordre *d'un schéma comme l'ordre maximal de ses non-terminaux.*

---

10. On se contentera de les appeler *schémas* dans la suite.
11. Il s'agit donc de PCF avec constantes non-interprétées d'ordre 1.
12. On utilisera des lettres majuscules pour les non-terminaux, et des minuscules pour les terminaux.

Un schéma $G$ définit un terme (et un seul, puisqu'il n'y a qu'une règle par non-terminal), et donc un arbre, noté $[\![G]\!]$. On l'obtient par l'application des règles de $G$ à partir du symbole initial $S$, cette application étant éventuellement infinie. On considèrera aussi dans la dernière partie le *graphe* généré par un schéma. Celui-ci s'obtient en prenant tout d'abord la forêt constituée des termes apparaissant à droite dans une règle du schéma, puis en ajoutant pour chaque occurence d'un non-terminal dans l'un des arbres de la forêt une arête dirigée de ce non-terminal vers la racine de l'arbre représentant le terme apparaissant à droite dans la règle de réécriture lui correspondant. Les chemins sur ce graphe et sur $[\![G]\!]$ sont en correspondance, puisque $[\![G]\!]$ est le *déploiement* de ce graphe. Considérer ces deux productions équivalentes du schéma permettra à la fois d'utiliser le graphe, toujours fini, pour créer un jeu de vérification, alors qu'on pourra exécuter un automate d'arbre sur $[\![G]\!]$.

### 2.2. *Transformation RHS et explicitation de la dynamique sous-jascente aux schémas*

Là où la dynamique d'un schéma usuel est purement syntaxique, on peut, par une transformation adaptée, lui associer un schéma équivalent (dans un sens à préciser) produisant un terme du $\lambda$-calcul (avec marquage explicite des applications) et dont la dynamique est donnée par la règle de $\beta$-réduction.

**Définition 14 (Transformation RHS)** *La transformation RHS du schéma G est donnée par un processus de réécriture de ses règles en quatre étapes ; elle produit un nouveau schéma $\bar{G}$ :*

*1) On réalise l'$\eta$-expansion de chaque sous-terme apparaissant en position d'argument d'une application, même s'il est de type de base.*

*2) On insère un symbole explicite d'application @ : toute application $De_1 \cdots e_n$ se réécrit $@De_1 \cdots e_n$.*

*3) On curryfie les règles : $F\xi_1 \cdots \xi_n \to e$ devient $F \to \lambda\xi_1 \cdots \xi_n.e$ (même si $n = 0$, auquel cas ceci revient à rajouter $\lambda.$ en tête de la partie droite de la règle).*

*4) On $\alpha$-renomme les variables liées, de sorte que deux variables de lieur différent aient un nom différent.*

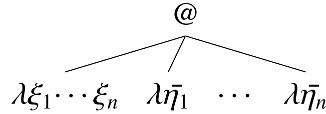
$\bar{G}$ génère l'*arbre de calcul* du schéma original $G$, noté $\lambda(G)$. Il s'agit en fait de l'arbre de Böhm $\eta$-long de $[\![G]\!]$. On définit une notion de *direction* sur $\lambda(G)$ : le fils le plus à gauche d'un nœud @ aura pour direction $0$, son voisin la direction $1$, et ainsi de suite ; pour un nœud parent étiqueté différemment, on procèdera de même en commençant le compte des directions à $1$. On note Dir($f$) l'ensemble des directions affectées aux fils d'un symbole donné $f$.

Maintenant que l'on a associé à un schéma $G$ un schéma $\bar{G}$ révélant son comportement dynamique en termes de $\beta$-réduction, on peut comprendre intuitivement ce qu'est l'ordre d'un schéma : il s'agit de la quantité maximale de $\beta$-réductions imbriquées à effectuer lors de la réduction du terme généré par $\bar{G}$, à laquelle on retranche un, puisqu'un processus de substitution similaire à celui de la $\beta$-réduction (mais sans aller en profondeur) a lieu au cours de la dérivation donnant $\lambda(G)$. Notons que l'ordre de $\bar{G}$ est $0$ : il génère un arbre *régulier* - la complexité de la $\beta$-réduction apparaîtra ailleurs, dans les *traversées*. Intuitivement, l'ordre d'un schéma décrit la complexité de l'arbre qu'il induit.

2.2.1. *Structure locale de* $\lambda(G)$.

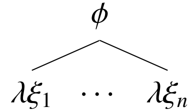
La transformation RHS donne à l'arbre généré $\lambda(G)$ une structure très agréable. En effet, la forme locale de celui-ci est la suivante :

– Un nœud applicatif (étiqueté @) d'arité $n$ a $n+1$ fils :



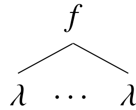

où tout fils de direction $i > 0$ est la racine du terme correspondant au $i$-ème argument de l'application, et où le fils le plus à gauche, de direction $0$, correspond à la racine du terme auquel sont appliqués ces arguments. $\xi_i$ est la variable correpondant au $i$-ème argument de l'application : ainsi, lors d'une $\beta$-réduction de cette application selon une stratégie en *appel par nom*, on commencera par explorer le sous-arbre en direction $0$, puis on remplacera toute occurence de $\xi_i$ par le terme obtenu par évaluation du terme en direction $i$. Comme le schéma de récursion d'ordre *supérieur* peut générer un terme d'ordre 2 ou plus, le terme à substituer à $\xi_i$ peut à son tour devoir être réduit : les variables correspondant à ses arguments sont données par la liste (potentiellement vide) $\bar{\eta}_i$.

– Un nœud étiqueté par une variable $\phi$ d'arité $n$ a $n$ fils, de directions $1$ à $n$ :




où la $i$-ème direction est celle menant au terme apparaissant comme $i$-ème argument de $\Phi$. Si $n = 0$, le nœud est terminal.

– Un nœud étiqueté par une élément de $\Sigma$ d'arité $n$ a $n$ fils :




Si $n = 0$, le nœud est terminal.

**Exemple 4 (De $G$ à $\lambda(G)$)** *Considérons le schéma G d'ordre 2 suivant :*

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

*La transformation RHS donne $\bar{G}$ :*

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

*La Figure 1 montre les arbres $[\![G]\!]$ et $\lambda(G)$. On remarquera qu'il n'y a* a priori *aucun lien entre les chemins dans ces deux arbres. Ceci peut sembler déroutant, mais provient en fait du fait que là où G est un schéma d'ordre $n$, et que le processus de dérivation de $[\![G]\!]$ correspond donc à un processus de substitution équivalent à une $\beta$-réduction totale du terme sous-jascent, $\bar{G}$ n'est qu'un schéma d'ordre 0 : la substitution n'a pas lieu en profondeur, et reste à effectuer sur le terme $\lambda(G)$ par $\beta$-réduction.*

### 2.3. $\bar{G}$ *calcule bien* $[\![G]\!]$ *: la correspondance chemins-traversées*

2.3.1. *Définition informelle des traversées.*

Quand on transforme $G$ en $\bar{G}$, on obtient un schéma (d'ordre 0) générant un terme de $\lambda$-calcul non évalué explicitant la dynamique de réécriture du schéma (d'ordre $n$), en termes de $\beta$-réduction. On va utiliser sur ce terme des techniques de sémantique des jeux ; la structure alternante[13] de $\lambda(G)$ entre des nœuds étiquetés par un $\lambda$ et d'autres ne l'étant pas y incite : attribuons donc les nœuds avec un $\lambda$ à l'environnement, et les autres au programme. Cependant, il va falloir aller un peu plus loin dans l'analyse de la réduction (donc, de l'exécution) du terme que dans la section précédente : en effet, le terme $\lambda(G)$ est par construction de type $o$ ; son interprétation serait donc simplement une stratégie sur l'arène $o$, composée d'une question de l'environnement ("Terme, quelle forme normale donne ta $\beta$-réduction ?"), et d'une réponse du programme : sa forme normale. Or le terme contient des sous-termes appliqués à d'autres : on peut donc voir son exécution comme obtenue par application de ces sous-termes les uns aux autres, ceci étant obtenu par composition des stratégies correspondantes sur leurs arènes de types. C'est l'opération d'effacement de l'interaction dans l'arène intermédiaire qui cache les opérations intermédiaires de calcul, mais celles-ci nous intéressent

13. Ceci explique pourquoi certaines opérations apparemment inutiles ont eu lieu au cours de la transformation RHS, par exemple, la curryfication des règles correspondant à des non-terminaux de type de base $o$.

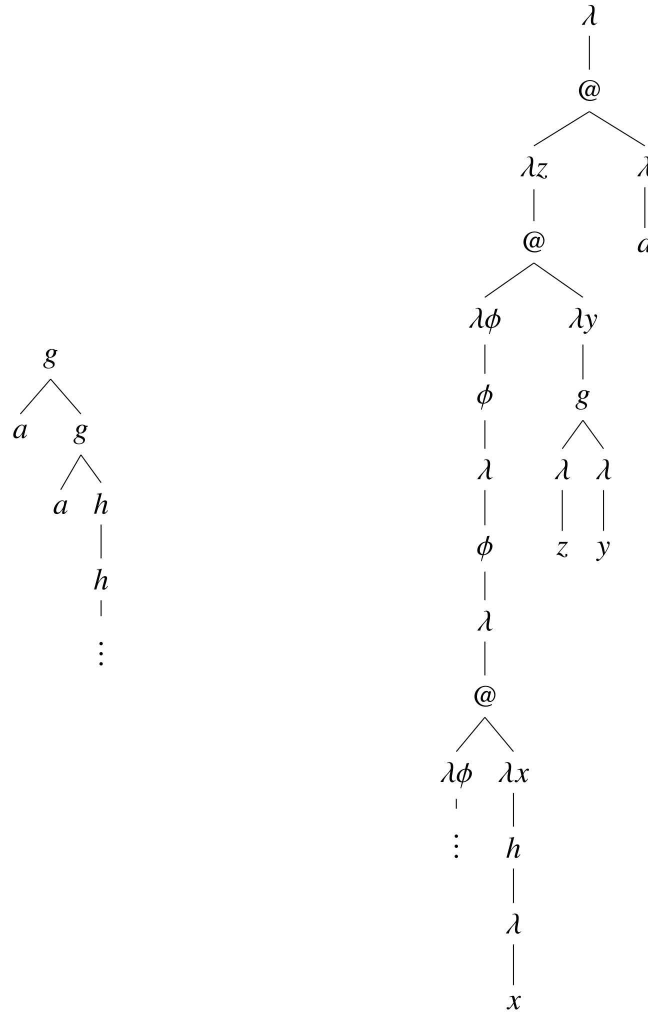



**Figure 1.** $[\![G]\!]$ *et* $\lambda(G)$

tout particulièrement. On s'intéresse donc à une *sémantique des jeux révélée*, dans laquelle on ne cache plus l'interaction lors de la composition[14], de sorte que l'on va obtenir des traces d'exécution intéressantes même si $\lambda(G)$ est de type $o$. L'idée de cette révélation totale fut introduite dans (Ghica *et al.*, 2005) et (Greenland, 2004) ; nous

14. Il est trivial de cacher l'interaction lors de la composition ; mais il est plus intéressant de savoir que la reconstruction de la suite d'interaction cachée lors de la composition peut être reconstituée, voir (Hyland *et al.*, 2000, Partie II).

utiliserons ici une révélation moindre (et plus proche de l'intuition), introduite dans (Blum *et al.*, 2008) et développée dans (Blum *et al.*, 2011). L'idée est donc que $\lambda(G)$ peut s'interpréter sur une arène composée à partir de celles des types apparaissant dans ses sous-termes ; les arènes donnant ces types étant obtenues par les constructions de la section précédente à partir de l'arène du type simple $o$ : l'application d'un terme à plusieurs arguments correspond à prendre le tenseur des arènes des types de ces arguments, puis on prend l'exponentielle de ce produit tensoriel (les arguments pourraient devoir être lus plusieurs fois dans une stratégie en appel par nom), et on crée l'application de cette arène vers celle du type du résultat du terme auquel on applique les arguments. La structure de cette arène est très proche de celle de $\lambda(G)$ - il "suffit" d'enlever les nœuds applicatifs $@$, qui n'ont pas de sens en $\lambda$-calcul, mais permettent d'encoder les applications dans $\lambda(G)$, et d'identifier correctement des sommets pour remédier au fait qu'un simple effacement des $@$ enlève la propriété d'alternance des polarités des sommets nécessaire à l'obtention d'une arène, voir (Blum *et al.*, 2011).
L'arène est de fait potentiellement infinie, puisqu'on la calque sur $\lambda(G)$ : cependant, tout comme il y a équivalence entre le graphe fini $Gr(G)$ et l'arbre $\lambda(G)$, on peut, au lieu de *déployer* les récursions dans le schéma, autoriser à rejouer (à l'aide de l'exponentielle) sur l'arène qui aurait été déployée par le processus décrit ci-dessus.
On va maintenant pouvoir définir, sur l'arène révélée donnée par $\lambda(G)$, des stratégies effectuant la $\beta$-réduction en appel par nom[15] des branches de $\lambda(G)$ ; on verra ensuite que celles-ci sont en correspondance avec les branches de l'arbre original $[\![G]\!]$. Ces stratégies seront appelées *traversées*, on va en effet voir qu'elles "sautent" parfois dans l'arbre $\lambda(G)$, pour aller chercher un argument donné quand cela est nécessaire. Le choix de la branche à calculer est effectué par l'environnement : comme il choisit les coups correspondant aux nœuds avec un $\lambda$, c'est son tour de jouer quand on est sur un nœud étiqueté par un symbole de $\Sigma$, et c'est donc lui qui choisit la direction de l'arbre $[\![G]\!]$ à calculer.

2.3.2. *Définition formelle des traversées.*

Il faut tout d'abord définir une relation d'activation (on rajoute ici un entier à l'activation, ce qui rendra plus commode le contrôle de la $\beta$-réduction) sur les sommets de $\lambda(G)$, donnant la forme de l'arène sous-jascente :

**Définition 15 (Relation d'activation)** *Sur les noeuds de $\lambda(G)$, on définit l'activation comme suit :*

*– Tout nœud étiqueté par un $\lambda$ et différent de la racine est $i$-activé par son nœud parent, où $i$ est sa direction depuis ce dernier.*

*– Tout nœud étiqueté par une variable est $i$-activé par son lieur $\lambda\bar{\xi}$ (avec $\xi_i$ le $i$-ème élément de la liste $\bar{\xi}$).*

15. Il ne saurait s'agir d'un appel par valeur, puisqu'on parcourt simplement un arbre : on peut retourner lire plusieurs fois la valeur d'un argument grâce à la construction exponentielle, mais pas modifier la sous-arène codant sa valeur pour la retenir. On pourrait éventuellement définir des stratégies à mémoire, mais on y perdrait l'*innocence*.

*Dans ce cadre, une* suite justifiée *est juste une suite de nœuds de* $\lambda(G)$ *alternant les sommets étiquetés ou non par un* $\lambda$*, et tel qu'il n'y ait aucun nœud* $i$*-activé par un nœud n'apparaissant pas dans la suite justifiée.*

On représentera encore les suites justifiées avec des pointeurs, mais on leur ajoutera l'entier $i$. On rappelle la notion de vue pour le programme, avec cette variante de pointeurs de justification étiquetés :

**Définition 16 (Vue d'une suite justifiée)** *La vue* $\lceil t \rceil$ *d'une suite justifiée* $t$ *se définit par induction :*

– $\lceil \lambda \rceil = \lambda$

– $\lceil t n \ldots \lambda\bar{\xi} \rceil = \lceil t \rceil n \quad \lambda\bar{\xi}$ (pointeur étiqueté $i$ de $\lambda\bar{\xi}$ vers $n$)

– $\lceil t\lambda\bar{\xi}n \rceil = \lceil t\lambda\bar{\xi} \rceil n$

*où dans la dernière règle un pointeur de justification est ajouté à la vue si* $n$ *en avait un et qu'il avait pour but un élément de la vue.*

**Définition 17 (Traversée)** *Une traversée est une suite justifiée générée incrémentalement par les cinq règles suivantes :*

*1) La suite justifiée composée de la racine de* $\lambda(G)$ *est une traversée.*

*2) Si* $t$ @ *est une traversée, alors* $t \ @ \ \lambda\bar{\phi}$ (pointeur étiqueté $0$ de $\lambda\bar{\phi}$ vers @) *en est également une.*

*3) Si* $t\ f$ *(où* $f \in \Sigma$*) est une traversée,* $t\ f\ \lambda$ (pointeur étiqueté $i$ de $\lambda$ vers $f$) *en est également une pour tout* $1 \leq f \leq ar(f)$*.*

*4) Si* $t\ n\ \lambda\bar{\xi} \cdots \xi_i$ (pointeur étiqueté $i$ de $\xi_i$ vers $\lambda\bar{\xi}$) *est une traversée,* $t\ n\ \lambda\bar{\xi} \cdots \xi_i\ \lambda\bar{\eta}$ (pointeurs étiquetés $i$ de $\xi_i$ vers $\lambda\bar{\xi}$ et de $\lambda\bar{\eta}$ vers $n$) *en est une aussi*[16]*.*

*5) Si* $t\lambda\bar{\xi}$ *est une traversée et que* $\lceil t\lambda\bar{\xi}n \rceil$ *est un chemin dans* $\lambda(G)$*, alors* $t\lambda\bar{\xi}n$ *est aussi une traversée.*

Explicitons cette définition :

– La première règle initialise l'évaluation du terme à la racine de $\lambda(G)$.

– La deuxième règle explique que lorsqu'on rencontre une application, on commence par en évaluer la partie gauche et pas les arguments - la stratégie de réduction est en appel par nom.

16. C'est ici qu'apparaît l'intéret de l'étiquetage des pointeurs de justification : ils permettent de définir aisément cette règle, assurant que lors d'une $\beta$-réduction faisant appel au calcul et à la substitution du $i$-ème argument, on "saute" bien dans l'arbre à la racine du sous-arbre décrivant ce $i$-ème argument.

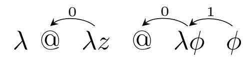
– La troisième règle explique que lorsqu'on rencontre un symbole de l'alphabet de l'arbre original toutes les directions prises ensuite sont valides - en effet, ici le choix n'est pas laissé au programme, mais à l'environnement ; le programme doit donc considérer toutes les possibilités dans sa stratégie.

– La quatrième règle explique que lorsqu'on rencontre la variable désignant le $i$-ème paramètre d'une application, on va évaluer celui-ci, en allant à la position qui lui correspond dans l'arbre. L'$\alpha$-renommage effectué au cours de la transformation RHS assure l'absence d'ambiguïté sur les sauts.

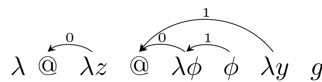
– La cinquième règle permet de gérer le cas des nœuds étiquetés par un symbole de $\Sigma$ ou un @, qui ne font pas réellement partie de l'arène révélée ; sur ceux-ci, on se contente de poursuivre la descente dans la branche que l'on est en train de visiter.

Notons que les traversées décrivent en fait à la fois l'ensemble des stratégies du programme et celles de l'environnement : pour retomber sur la notion usuelle de stratégie pour le programme, il faudrait exclure de l'ensemble des traversées les suites justifiées construites à l'aide de ces règles mais de longueur impaire.

**Exemple 5 (Traversées de $\lambda(G)$)** *On reprend le schéma de l'exemple précédent. Pour calculer ses traversées, on doit commencer de la racine (règle 1), puis on descend dans l'arbre (règle 5) en prenant la direction 0 quand on trouve un symbole applicatif @ (règle 2), ce qui nous amène à :*

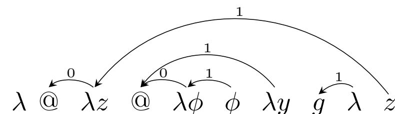


*La $\beta$-réduction correspond alors à remplacer $\phi$ par sa valeur ; or la structure de $\lambda(G)$ assure que celle-ci est donnée par (la $\beta$-réduction de) l'argument correspondant au sous-arbre partant en direction 1 du second symbole @ recontré jusqu'ici, il faut donc poursuivre en visitant le nœud $\lambda y$, et c'est ce qu'impose la quatrième règle. La cinquième prolonge l'exploration de la branche courante, et on arrive à la traversée suivante :*

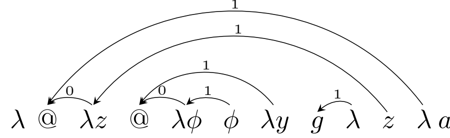


*L'idée est qu'on vient de recontrer le symbole $g$ correspondant à la racine de $[\![G]\!]$ ; l'environnement a maintenant le choix de la direction à calculer (règle 3) :*

*– S'il demande de calculer la direction* 1*, la règle 5 amène à la traversée suivante :*

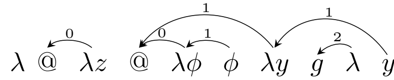


*et $z$ correspond à l'argument du premier symbole @, dont on va donc chercher à calculer la valeur :*

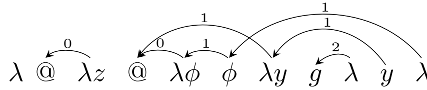


*et cette traversée est maximale, puisque sur $a$ seul la cinquième règle s'applique ; or $a$ est d'arité nulle. Remarquons que cette traversée maximale correspond à la branche $g \cdot a$ de $[\![G]\!]$ lorsqu'on la projette sur $\Sigma$.*

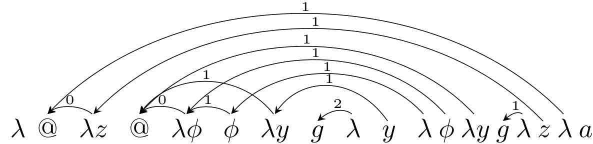
*– S'il demande de calculer la direction $2$, la règle 5 amène à la traversée suivante :*



*et $y$ est défini pour référer à l'argument de $\phi$, que la traversée va donc aller évaluer :*



*avec $\lambda$ le fils de $\phi$. La cinquième règle amène à visiter $\phi$ à nouveau, on retourne donc visiter le sous-arbre lui correspondant ; si l'environnement choisit la première direction lors de la visite de $g$, on obtient :*



*qui est à nouveau maximale, et correspond à la branche $g \cdot g \cdot a$ de $[\![G]\!]$ ; tandis que si l'environnement choisit en $g$ de visiter la seconde direction, on entre dans un processus de réduction infini où $h$ est visité infiniment souvent, ce qui correspond à la branche infinie $g \cdot g \cdot h \cdot h \cdots$ de $[\![G]\!]$.*

*Pour un exemple à l'ordre 3, où des sauts de $\beta$-réduction peuvent avoir lieu à l'intérieur de tels sauts, on pourra consulter (Grellois, 2010, Annexe A), ou alors utiliser le logiciel HOG de William Blum (`http://william.famille-blum.org/research/tools.html`), qui permet de générer l'arbre de calcul d'un schéma et de construire ses traversées. De nombreux exemples sont fournis.*

On peut pour finir se demander quelles sont les propriétés des stratégies correspondant aux traversées. Intuitivement, comme elles représentent des termes de PCF, elles devraient être innocentes et bien parenthées. Le bon parenthésage provient de la forme générale des traversées (Ong, 2006, Proposition 6) ; quant à l'innocence, elle est immédiate au vu de la définition des traversées : le joueur ne peut jouer que selon la cinquième règle, les quatre autres décrivant implicitement le comportement de la

stratégie de l'environnement ; or cette cinquième règle n'utilise que la vue, donc la stratégie du programme est innocente.

2.3.3. *Correspondance chemins-traversées*

Les traversées correspondent (il y a isomorphisme par une opération de projection convenable, visant à enlever les symboles @ qui n'ont pas de réel intérêt sémantique) aux traces en sémantique des jeux révélée de la stratégie de $\beta$-réduction en appel par nom sur (l'arène révélée induite par) $\lambda(G)$ ((Blum *et al.*, 2008, Théorème 2.1),(Blum *et al.*, 2011, Théorème 2.2)). On a plus particulièrement le théorème suivant (Ong, 2006, Théorème 7) :

**Théorème 1** *Il y a bijection entre les chemins maximaux de l'arbre* $[\![G]\!]$ *et les traversées maximales de* $\lambda(G)$ *; et la projection sur* $\Sigma$ *d'une traversée maximale donne exactement l'étiquetage du chemin maximal correspondant dans* $[\![G]\!]$.

## 3. $\mu$-calcul modal, jeux de vérification et traversées

Maintenant que nous avons une notion de *modèle* pour les programmes à vérifier, il nous faut un moyen logique d'exprimer les propriétés à vérifier sur ses productions. Comme ce sont des arbres, on va considérer une logique adaptée, le *$\mu$-calcul modal*, puis nous verrons qu'un modèle bien particulier d'automates d'arbres lui est équivalent et amène à des jeux de vérification. Enfin, nous verrons que la correspondance chemins-traversées nous permet de mêler le travail d'explicitation sémantique du contenu des schémas mené jusqu'ici à notre tâche de vérification.

### 3.1. *$\mu$-calcul modal*

3.1.1. *Définition informelle*

Le $\mu$-calcul modal est une logique sur les structures à branches. Elle ajoute aux connecteurs booléens usuels deux opérateurs de point fixe $\mu$ (plus petit point fixe) et $\nu$ (plus grand point fixe), ainsi que deux modalités sur la structure à branches $\Box$, demandant qu'une formule soit vraie pour tous les successeurs *immédiats* du nœud courant, et $\diamond$, ne l'exigeant que pour un successeur immédiat (au moins). Ces deux modalités correspondent donc aux quantifications universelles et existentielles sur l'ensemble des successeurs immédiats d'une position. Il peut sembler fort restrictif de se cantonner à la quantification sur les successeurs immédiats ; cependant, l'introduction des opérateurs de point fixe $\mu$ et $\nu$ permet d'obtenir une notion de récursion dans les formules. Dans la mesure où les structures à branches considérées ici sont des arbres (que l'on oriente de la racine vers les feuilles), on peut affiner leur interprétation : $\mu$ permet la récursion strictement *finie* sur une formule, alors que $\nu$ autorise à boucler

infiniment[17]. Ainsi, on peut utiliser ces deux opérateurs pour caractériser des propriétés telles que “il existe un $a$ sur une branche” ou “il n'y a pas de $b$ dans l'arbre”, comme nous allons le voir sur des exemples. Cette étude se contentera d'un traitement assez informel du $\mu$-calcul modal et de sa correspondance avec les automates alternants à parité, pour un traitement plus rigoureux on pourra consulter (Bradfield *et al.*, 2001), ou alors (Wilke, 2002) qui utilise une définition des automates alternants à parité équivalente à celle que nous verrons et aidant à l'intuition de la correspondance avec le $\mu$-calcul modal.

**Exemple 6 (Quelques formules de $\mu$-calcul modal sur un arbre)**

– $\mu Z.(\Box Z)$ *est vraie si et seulement si l'arbre est fini : tout d'abord, on vérifie à la racine la formule* $\Box Z$*, or celle-ci exige de vérifier sur tous les fils de la racine la formule* $Z$*. Mais, par point fixe,* $Z = \Box Z$*, donc sur chacun de ces fils, on vérifie* $Z$ *sur les fils, et ainsi de suite. Au final, ce processus de vérification termine si et seulement si l'arbre est fini, auquel cas la formule est vraie - s'il y a une branche infinie, la récursion ne termine pas, et ceci est autorisé pour une récursion donnée par* $\nu$ *mais pas par* $\mu$*.*

– $\mu Z.(a \vee \diamond Z)$ *est vraie si et seulement si une branche de l'arbre contient un* $a$ *: on vérifie* $a \vee \diamond Z$ *à la racine pour commencer ; si la formule est vraie, soit il y a un* $a$ *à la racine, soit* $\diamond Z$ *est vraie, c'est à dire qu'il existe un fils de la racine sur lequel* $Z = a \vee \diamond Z$ *(par point fixe) est vérifiée : soit il y a un* $a$ *sur ce fils, soit on itère sur un de ses fils, . . . Comme la récursion doit être finie (l'opérateur de récursion est* $\mu$*), l'itération doit stopper : ceci implique qu'il y ait un* $a$ *à une distance finie de la racine, donc sur une des branches de l'arbre. Si on avait choisi un* $\nu$*, la propriété aurait été “il y a un* $a$ *sur une branche ou une branche infinie”.*

– *Comment caractériser l'existence d'une branche ne contenant que des* $h$ *après un nombre fini de sommets quelconques, comme sur le* $[\![G]\!]$ *de l'exemple 4 ? Il faut exprimer le fait qu'après un certain chemin (fini) une formule donnée est vraie, puis remplacer cette formule par celle décrivant une branche avec seulement des* $h$*. Pour dire que* $\phi$ *est vraie après un chemin fini, il devrait être clair que la formule est* $\mu Z.(\phi \vee \diamond Z)$*. Pour exprimer l'existence d'une branche infinie avec uniquement des* $h$*, on prend* $\phi = \nu Y.(h \wedge \diamond Y)$ *: cette fois, on veut à la fois qu'il y ait un* $h$ *et que* $Y = h \wedge \diamond Y$ *soit vraie sur un fils du nœud courant, et ce potentiellement à l'infini. Si on avait voulu exprimer l'existence d'une branche infinie ne contenant que des* $h$ *(après un chemin fini de sommets quelconques), il aurait fallu prendre* $\phi = \nu Y.(h \wedge \diamond Y) \wedge \neg(\mu Z.(h \wedge \diamond Z))$ *: on veut avoir une branche ne contenant que des* $h$*, potentiellement infinie (opérateur* $\nu$*) mais pas finie (négation de l'opérateur* $\mu$*).*

REMARQUE. — CTL se traduit en $\mu$-calcul modal
La seule difficulté est de traduire les deux opérateurs $\forall[\phi \mathbf{U} \psi]$ et $\exists[\phi \mathbf{U} \psi]$. Le premier exprime le fait que pour tout chemin infini commençant au nœud courant, $\phi$ est vraie jusqu'à ce qu'après un nombre fini de nœuds on en atteigne un où $\psi$ est vraie. Le

17. Voir (Bradfield *et al.*, 2001, Sections 3.1 et 3.2).

second modélise l'existence d'un chemin fini commençant du nœud courant sur lequel $\phi$ est vérifiée jusqu'à ce qu'un nœud satisfaisant $\psi$ soit rencontré.
Dans le premier cas, il suffit d'itérer finiment la formule $\psi \vee \Box\phi$, ce qui nous donne $\mu Z.(\psi \vee (\Box\phi \wedge Z))$ ; dans le second, la quantification sur les chemins est existentielle, il suffit donc de remplacer $\Box$ par $\diamond$ et on obtient $\mu Z.(\psi \vee (\diamond\phi \wedge Z))$. Pour une traduction plus formelle, voir (Bradfield *et al.*, 2001, Section 4.1).

### 3.2. *Vérification d'une formule de $\mu$-calcul modal sur un modèle générant des arbres à l'aide d'automates alternants à parité*

#### 3.2.1. *Automates alternants à parité*

Les automates alternants à parité généralisent le comportement des automates d'arbres non-déterministes, nous allons voir que leur comportement calculatoire est particulièrement adapté à la vérification de formules de $\mu$-calcul modal sur des arbres.

**Définition 18 (Automates alternants à parité)** *Un automate alternant à parité*[18] *est un quintuplet* $< \Sigma, Q, \delta, q_0, \Omega >$ *où :*

– $\Sigma$ *est un alphabet fini (avec une notion d'arité) dont les éléments sont les étiquettes des nœuds des arbres d'entrée,*

– $Q$ *est l'ensemble fini des états de l'automate,*

– $\delta$ *est la fonction de transition de l'automate. Elle envoie tout couple* $(q, f) \in Q \times \Sigma$ *sur une formule booléenne positive*[19] *dont les atomes sont des couples de* $\mathrm{Dir}(f) \times Q$*,*

– $q_0$ *est l'état initial,*

– $\Omega : Q \to \mathbb{N}$ *est la fonction de priorité des états de l'automate.*

Ceci généralise le déterminisme, au sens suivant : au lieu d'avoir deux transitions possibles $\delta(q, f) = (q_1, \cdots, q_n)$ et $\delta(q, f) = (q'_1, \cdots, q'_n)$ partant d'une même configuration $(q, f)$, on aura dans un automate alternant :

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

**Définition 19 (Arbre d'exécution d'un automate alternant)** *Un arbre d'exécution d'un automate alternant* $\mathcal{A}$ *sur une entrée* $t$ *étiquetée par* $\Sigma$ *est un arbre* $r$ *(sans notion d'arité) tel que :*

– $\epsilon \in \mathrm{Dom}(r)$ *et* $r(\epsilon) = (\epsilon, q_0)$

18. On se contentera dans la suite d'*automate alternant*.
19. Ceci n'est pas une restriction : dans la mesure où $\mathrm{Dir}(f) \times Q$ est fini, on peut repousser toutes les négations sur les atomes par les lois de Morgan puis remplacer les atomes niés par leur complémentaire dans $\mathrm{Dir}(f) \times Q$ pour simuler une négation.

– *Pour tout* $\beta \in \mathrm{Dom}(r)$ *avec* $r(\beta) = (\alpha, q)$ *et* $\delta(q, t(\alpha)) = \theta$*, il y a un ensemble* $S$ *vérifiant* $\theta$ *et tel que pour tout* $(a, q') \in S$*, il y a une direction* $b$ *sur* $r$ *telle que* $r(\beta b) = (\alpha a, q')$.

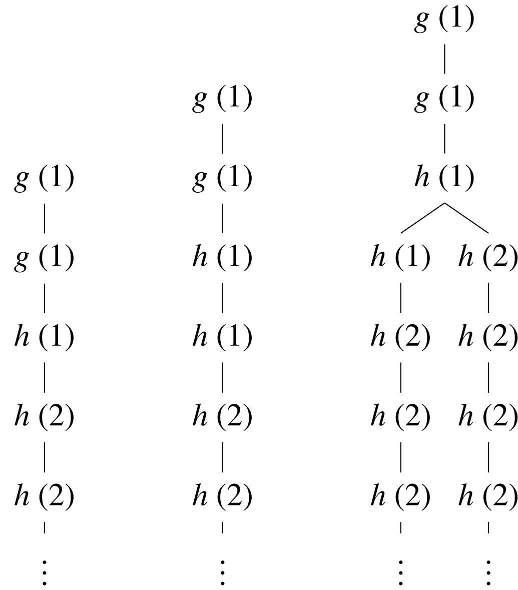
*On dit qu'un arbre d'exécution* $r$ *de* $\mathcal{A}$ *sur* $t$ *est* acceptant *si le jeu de parité sous-jascent est gagnant pour Eve : ceci signifie que sur toute branche infinie de* $r$ *la priorité minimale parmi les priorités rencontrées infiniment souvent doit être paire.*

Il n'y a donc pas de correspondance entre les nœuds de $r$ et de $t$ : un nœud de $t$ pourrait apparaître plusieurs fois (ou même aucune) dans $r$, étiqueté par des états différents - c'est ce qui fait que $r$ n'a pas de notion d'arité. Ainsi, $r$ peut contenir plusieurs exécutions parallèles de calcul sur des sous-arbres : ce sera très pratique pour vérifier des conjonctions logiques sur $t$.

**Exemple 7 (Arbre d'exécution d'un automate alternant)** *On considère un automate alternant sur l'alphabet avec arité* $\{g : 2\,;\, h : 1\,;\, a : 0\}$*, d'ensemble d'états* $\{1, 2\}$ *et dont la fonction de priorité envoie chaque état sur la priorité entière donnée par son nom. L'état initial est* $1$*, et on définit* $\delta$ *comme suit :*

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

*avec* $\delta = \perp$ *sur les autres couples de* $Q \times \Sigma$*. Si on reprend le schéma* $G$ *de l'exemple 4, les arbres suivants sont des arbres d'exécution de cet automate alternant sur* $[\![G]\!]$ *(on a représenté côte à côte le symbole de l'arbre d'entrée et l'état étiqueté par l'automate alternant) :*




*Tous ces arbres d'exécution sont acceptants : la seule priorité rencontrée infiniment souvent est* 2*, et* 2 *est pair.*
*Remarquons que* $[\![G]\!]$ *n'est pas un arbre d'exécution de cet automate alternant ; c'en serait un si l'on ajoutait la transition* $\delta(1, a) = \top$.

3.2.2. *Vérification d'une formule de* $\mu$*-calcul modal sur un arbre à l'aide d'un automate alternant à parité*

Emerson et Jutla (Emerson *et al.*, 1991) ont prouvé le résultat suivant :

**Théorème 2** *Il est équivalent de vérifier la véracité d'une formule de* $\mu$*-calcul modal* $\phi$ *à la racine d'un arbre* $t$ *et l'existence d'un arbre d'exécution sur* $t$ *pour un automate alternant à parité* $\mathcal{B}_\phi$.

Nous donnerons simplement ici une intuition de ce résultat et de la construction de $\mathcal{B}_\phi$, on verra (Wilke, 2002, Section 2.3) pour une preuve formelle. L'idée est que la formule de $\mu$-calcul modal $\phi$ à vérifier à la racine d'un arbre $t$ peut l'être par un automate alternant à parité : déjà, on élimine les négations en les repoussant sur les atomes par les lois de Morgan, puis on remplace les négations sur les atomes (de $\phi$), qui sont des éléments de l'alphabet fini $\Sigma$, et qu'on peut donc remplacer par la disjonction (finie) des éléments formant leur complémentaire dans $\Sigma$. On obtient ainsi une formule $\bar{\phi}$ sans négations. L'idée est ensuite d'étiqueter un arbre d'exécution $r$ de $\mathcal{B}_\phi$ sur $t$ par les états correspondant aux sous-formules de $\bar{\phi}$ qu'on vérifie en un noeud donné. Si $\bar{\phi}$ est booléenne, c'est-à-dire sans points fixes ni modalités $\diamond$ ou $\Box$, il

suffit de la vérifier à la racine : donc $\delta$ est définie sur l'état initial comme valant vrai si son étiquette satisfait $\bar{\phi}$, faux sinon, et cela suffit. Sinon, tout état correspond à la vérification d'une sous-formule $\psi$. A la racine, il correspond à $\bar{\phi}$, ce qui nous donne l'état initial. Dans un état donné, $\delta$ lit donc un symbole $f$, qui est une étiquette de l'arbre.

1) Si $\psi$ est booléenne, $\delta(q, f)$ vaut vrai si la sous-formule $\psi$ associée à $q$ est vraie au nœud courant, faux sinon.

2) Si $\psi = \diamond\eta$, $\delta(q, f) = \bigvee_{i\in \mathrm{Dir}(f)}(i, q_\eta)$, où $q_\eta$ est l'état associé à la vérification de $\eta$ : de sorte qu'un arbre d'exécution $r$ contiendra une (ou plusieurs) exécutions parallèles de la vérification de $\eta$ sur des fils de $f$.

3) Si $\psi = \Box\eta$, $\delta(q, f) = \bigwedge_{i\in \mathrm{Dir}(f)}(i, q_\eta)$ : $r$ doit contenir la vérification parallèle de $\eta$ sur tous les fils de $f$.

4) Si $\psi = \mu Z.\eta$, on remplace $\psi$ par $\eta[Z \leftarrow \psi]$ et on recommence ce processus sur cette nouvelle formule ; de même pour l'opérateur $\nu$.

5) Si $\psi = \eta_1 \vee \cdots \vee \eta_k \wedge \eta_{k+1} \wedge \cdots \wedge \eta_n$, on applique ces cinq règles aux $\eta_i$ (y compris celle-ci si nécessaire), ce qui nous donne une collection de formules booléennes positives $\zeta_i$ sur $\mathrm{Dir}(f) \times Q$ correspondant aux formules données par $\delta$ dans chaque cas ; on prend alors $\delta(q, f) = \zeta_1 \vee \cdots \vee \zeta_k \wedge \zeta_{k+1} \wedge \cdots \wedge \zeta_n$, en prenant soin de simplifier la formule si c'est possible (on verra dans l'exemple suivant qu'il peut arriver qu'elle soit toujours fausse - elle pourrait également être toujours vraie).

Reste à voir comment l'on traite les points fixes. Dans un premier temps, on s'est contenté de les itérer pour définir $\delta$ : les variables correspondant à des points fixes ont été remplacées par les formules auxquelles elles sont égales par l'équation de récursion (par exemple, si l'on doit traiter $\mu.Z(a \vee \diamond Z)$, on considèrera la formule itérée $\mu.Z(a \vee \diamond(a \vee \diamond Z))$ et c'est à elle qu'on réappliquera les étapes énumérées pour définir $\delta$ sur l'état courant). L'itération ayant eu lieu, on obtient un arbre d'exécution correspondant intuitivement à la vérification de $\phi$ sur $t$, à ceci près qu'on n'a pas encore vérifié que les formules dont l'itération est régie par un point fixe fini $\mu$ sont bien itérées un nombre fini de fois. C'est là qu'intervient la condition de parité discriminant les arbres d'exécution acceptants de ceux qui ne le sont pas : il est maintenant temps de définir la fonction de parité $\Omega$ de $\mathcal{B}_\phi$ de sorte qu'il n'existe un arbre d'exécution acceptant que si et seulement si l'arbre $t$ dont il représente l'exécution vérifie $\phi$. S'il n'y a pas d'opérateur de point fixe, il suffit de donner une priorité paire à tous les états, de sorte que tout arbre d'exécution est acceptant. Sinon, rappelons-nous que chaque état représente une sous-formule de $\phi$. Cette sous-formule peut n'être dans la portée d'aucun opérateur de point fixe, auquel cas on lui donnera une priorité arbitraire plus grande que toutes celles que nous allons donner maintenant. On ne considère maintenant plus ces sous-formules. Si dans $\phi$ l'opérateur le plus externe est un $\nu$, posons $i = 0$, et $i = 1$ si c'est un $\mu$. Toute sous-formule $\psi$ de $\phi$ considérée ici est dans la portée d'au moins un opérateur de récursion : on compte, dans la suite de ces opérateurs (du plus externe au plus interne dans $\phi$, et ayant à sa portée $\psi$), le nombre $n$ d'alternances entre des quantificateurs de type $\mu$ et de type $\nu$ ; la priorité de l'état correspondant à $\psi$ est alors $n + i$.

L'idée est simple : $\psi$ est itérée sur l'arbre par un certain nombre d'opérateurs de point fixe ; si un opérateur de type $\mu$ l'itère infiniment, alors un opérateur plus externe ne peut être à l'origine d'une itération infinie : donc toutes les priorités correspondant à des opérateurs plus externes, qui sont strictement inférieures, n'apparaissent qu'un nombre fini de fois sur les branches de $r$, de sorte que la priorité minimale parmi celles infiniment rencontrées correspond à un $\mu$, et notre construction fait qu'elle est impaire, de sorte que l'arbre d'exécution n'est pas acceptant.

**Exemple 8 (Un exemple d'automate alternant à parité associé à une formule)**
*Considérons l'arbre $[\![G]\!]$ de l'exemple 4, la formule $\phi$ de l'exemple 6 et l'automate alternant à parité de l'exemple 7. On a donc $\phi = \mu Z.(\psi \vee \diamond Z)$, où $\psi = \nu Y.(h \wedge \diamond Y)$. Il s'avère alors que l'automate alternant présenté dans l'exemple 7 vérifie si $\phi$ est vraie à la racine d'un arbre : il n'existe un arbre acceptant de cet automate sur $[\![G]\!]$ que si et seulement si celui-ci contient une branche ne contenant, à partir d'un certain stade, que des étiquettes $h$. Voyons comment le construire juste à partir de $\phi$. Il y aura deux états $1$ et $2$ comme on va le voir, le premier correspondant à vérifier $\phi$ (ou plutôt son itérée $\psi \vee \diamond\phi$) sur les nœuds qu'il étiquette, le second à $\psi$ (en fait, son itérée $h \wedge \diamond\psi$). L'état initial est $1$, puisque c'est $\phi$ qu'on veut vérifier à la racine. Pour ce qui est des priorités, $1$ vérifie $\phi$, qui est dans la portée (par point fixe) d'un opérateur $\mu$ et d'aucun autre opérateur de récursion, et est donc de priorité $0 + 1 = 1$ (0 alternances de type de quantificateur). $2$ correspond à $\psi$, qui est dans la formule totale $\phi$ dans la portée du $\mu$ et du $\nu$ : il y a donc une alternance de type de quantificateur, on a commencé à $1$ puisque le plus externe était un $\mu$, donc la priorité de $2$ est bien $2$.*

*– Sur l'état $1$ : par la règle 4, c'est en fait $\psi \vee \diamond\phi$ qu'on vérifie. Par la règle 5, on doit examiner séparément $\psi$ et $\diamond\phi$, puis on mettra dans $\delta(1, \cdot)$ ce qui correspondra à l'union des formules données par l'application itérée des cinq règles à $\psi$ et $\diamond\phi$.*

*- Pour $\diamond\phi$ : si le symbole courant est noté $x$, on a la formule booléenne positive sur $\mathrm{Dir}(x) \times Q$ donnée par $\bigvee_{i \in \mathrm{Dir}(x)} (i, 1)$ : l'état correspondant à la vérification de $\phi$ est $1$, et $\diamond$ amène à vérifier cette formule sur la disjonction des directions issues de $x$. Ceci donnera la formule vide sur $a$ (nullaire), $(1, 1)$ sur $h$ (unaire) et $(1, 1) \vee (2, 1)$ sur $g$ (binaire).*

*- Pour $\psi$ : la règle 4 amène à considérer à la place $h \wedge \diamond\psi$. On examine donc séparément les réécritures des formules $h$ et $\diamond\psi$ puis on en prend la conjonction. $h$ est vraie si et seulement si on lit $h$, quand à $\diamond\psi$, il se réécrit similairement à $\diamond\phi$. En prenant la conjonction, on obtient une formule fausse sur $(1, a)$ et $(1, g)$ et valant $(1, 2)$ sur $h$.*

*Quand on réunit tout ceci (en en prenant l'union), on obtient :*

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

*– Sur l'état $2$ : on veut vérifier $\psi$, on vient de montrer que ceci donne une formule*

*fausse sur* $(1, a)$ *et* $(1, g)$ *et valant* $(1, 2)$ *sur* $h$ *:*

- $\delta(2, a) = \bot$
- $\delta(2, g) = \bot$
- $\delta(2, h) = (1, 2)$

Si $[\![G]\!]$ contient une branche n'ayant que des $h$ après un certain temps, il existe un arbre d'exécution acceptant de cet automate alternant : notamment, n'importe quel arbre d'exécution de l'exemple 7 convient. Si la branche est finie, l'arbre est acceptant d'office, si elle est infinie, la seule priorité rencontrée infiniment souvent sur la branche est 2, donc paire, et l'arbre est acceptant. Réciproquement, il devrait être clair que l'existence d'un arbre d'exécution de cet automate sur un arbre implique l'existence sur celui-ci d'une branche ne contenant plus que des $h$ après un nombre fini de nœuds.

3.2.3. *Arbres de traversées*

Le problème de la vérification de la véracité d'une formule de $\mu$-calcul modal $\phi$ à la racine de $[\![G]\!]$ est donc équivalent au problème de l'existence d'un arbre d'exécution acceptant $r$ de $\mathcal{B}_\phi$ sur $[\![G]\!]$. La correspondance chemins-traversées permet d'aller plus loin : commençons par définir l'arbre des traversées d'un automate alternant sur un arbre de calcul $\lambda(G)$ :

**Définition 20 (Arbre des traversées d'un automate alternant sur $\lambda(G)$)** *Etant donnés un automate alternant à parité $\mathcal{B}_\phi$ et un arbre de calcul $\lambda(G)$, on définit un arbre de traversées $T_G$ de $\mathcal{B}_\phi$ sur $\lambda(G)$ comme suit :*

*– On prend un arbre d'exécution $r$ de $\mathcal{B}_\phi$ sur $[\![G]\!]$, tel qu'à chaque choix d'un ensemble de satisfiabilité $S \subset \mathrm{Dir}(f) \times Q$ dans la construction de $r$, $S$ ait été pris de cardinalité minimale parmi les ensembles satisfiant $\delta$.*

*– A chaque chemin maximal dans $r$ correspond*[20] *une traversée maximale de $\lambda(G)$ : $T_G$ est l'arbre obtenu de l'ensemble de ces traversées, commençant par le préfixe commun à toutes, puis branchant (sur un symbole de $\Sigma$) lorsqu'il n'y a plus de tel préfixe ; on poursuit la construction ainsi, de sorte que toute traversée maximale correspond à un chemin maximal de $T_G$.*

*– On réalise l'étiquetage de $T_G$ par des états de $\mathcal{B}_\phi$ : pour cela on parcourt l'arbre depuis la racine ; on étiquette d'abord tous les sommets rencontrés par l'état initial, puis lorsqu'on arrive à un symbole de $\Sigma$ on applique à ses fils les états donnés par l'étiquetage des fils de $r$ par $\mathcal{B}_\phi$. Sur chaque chemin, on propage cet étiquetage jusqu'à rencontrer un symbole de $\Sigma$, moment auquel on fait à nouveau correspondre l'étiquetage à celui de $r$, ...*

20. Il peut y avoir plusieurs copies d'une même branche de $[\![G]\!]$ dans $r$, pour vérifier un $\wedge$ ; dans ce cas, il y aura autant de copies de la traversée correspondante que de copies de la branche, simplement, les étiquetages par des états de $\mathcal{B}_\phi$ différeront.

Un arbre de traversées est *acceptant* s'il vérifie la condition de parité usuelle (sur toute branche la priorité minimale parmi celles rencontrées une infinité de fois est paire). On fait ainsi correspondre à tout arbre d'exécution acceptant de $\mathcal{B}_\phi$ sur $[\![G]\!]$ utilisant un minimum d'étiquetages pour satisfaire $\delta$ à chaque étape un arbre de traversées, qui est composé des traversées correspondant à chaque branche de $[\![G]\!]$, et dont la structure de branchement est donné par un regroupement par préfixes communs. Puisque dans l'arbre de traversées, l'état étiqueté ne dépend que des symboles de $\Sigma$, et que ceux-ci sont en correspondance avec ceux étiquetant les branches de $[\![G]\!]$, les conditions d'acceptance sont équivalentes.
Le problème de vérification de la véracité d'une formule de $\mu$-calcul modal $\phi$ à la racine de l'arbre généré par un schéma $G$ se réduit donc à celui de l'existence d'un arbre de traversées acceptant de $\mathcal{B}_\phi$ sur $\lambda(G)$.

**Exemple 9 (Arbres d'exécution et arbres de traversées)** *Considérons le schéma $G$ de l'exemple 4 et un automate alternant à parité $\mathcal{C}$ sur l'alphabet à arité $\{g : 2 ; h : 1 ; a : 0\}$, avec deux états $\{0,1\}$, où $1$ est l'état initial, et où le nom de chaque état indique sa priorité. On définit $\delta$ comme suit :*

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

*et on obtient ainsi l'arbre de traversées $\mathcal{C}$ sur $\lambda(G)$, représenté figure 2.*

### 3.3. *Vérification d'une formule de $\mu$-calcul modal sur un graphe*

Tout graphe fini définit un arbre, donné par son *déploiement* : il s'agit en fait de l'arbre donnant ses chemins en partant d'un sommet considéré comme initial, et dont la structure est donnée par les préfixes communs aux chemins. L'effet est en fait de "déplier" les cycles quitte à en ajouter des copies, ce qui donne une structure d'arbre (infini en présence de cycles). Pour vérifier une formule de $\mu$-calcul modal $\phi$ sur un graphe fini $G$, il suffit donc de vérifier l'existence d'un arbre d'exécution acceptant de l'automate alternant à parité associé $\mathcal{B}_\phi$ sur le déploiement de ce graphe.

## 4. Simulation des traversées

### 4.1. *Avec des automates alternants à parité*

Nous avons ramené le problème de la vérification de la véracité d'une formule de $\mu$-calcul modal $\phi$ à la racine de $[\![G]\!]$ à celui de l'existence d'un arbre de traversées acceptant de $\mathcal{B}_\phi$ sur $\lambda(G)$ ; nous allons maintenant construire un automate alternant à parité $\mathcal{C}$ dont les arbres d'exécution acceptants correspondront aux arbres de traversées

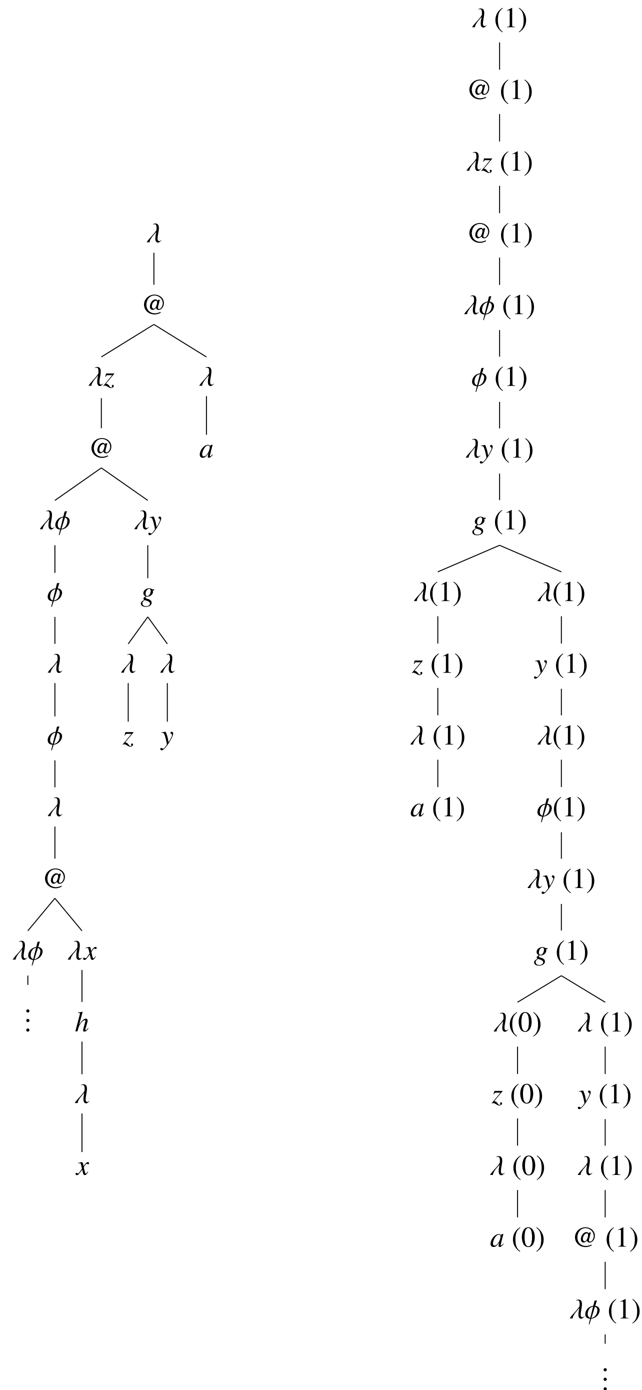
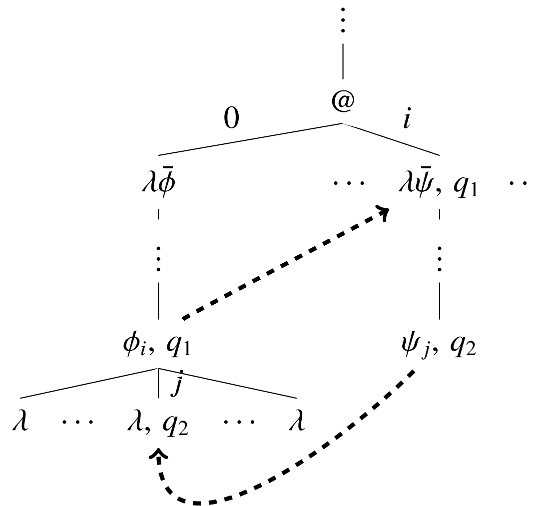



**Figure 2.** *$\lambda(G)$ et l'arbre de traversées pour $\mathcal{C}$ lui correspondant.*

acceptants de $\mathcal{B}_\phi$. On fixe maintenant $G$ (donc $[\![G]\!]$ et $\lambda(G)$) et $\mathcal{B}_\phi$, qu'on notera aussi $\mathcal{B}$.

4.1.1. *Description informelle de l'automate alternant simulant les traversées*

Afin de simuler les arbres de traversées sur $\lambda(G)$ avec un automate alternant à parité, il nous faut simuler proprement les sauts qu'effectue dans l'arbre une traversée, sachant qu'un tel automate est en substance un processus d'étiquetage de la racine vers les feuilles en une et une seule passe. La difficulté qui apparaît clairement ici - l'automate ne peut pas revenir en arrière dans l'arbre pour chercher l'information correspondant à un argument, ou communiquer entre ses exécutions sur des branches différentes - sera résolue à l'aide du non-déterminisme, et d'un système d'étiquetage assez complexe : on ne se contentera pas d'étiqueter $\lambda(G)$ avec les nœuds correspondant à l'automate de vérification $\mathcal{B}_\phi$, on ajoutera également aux annotations les valeurs de retour des évaluations d'arguments, devinées par non-déterminisme, à chaque fois qu'on rencontrera un symbole d'application $@$. Elles seront ensuite propagées dans la portée de l'application, avec un mécanisme de blocage en cas de mauvais pronostic, de sorte qu'un arbre d'exécution acceptant de $\mathcal{C}$ correspondra à un "sans-faute" dans les prédictions d'évaluation de la part de l'automate, simulant ainsi parfaitement le comportement attendu.
Prenons un exemple local : dans la situation suivante (d'ordre 2), le comportement attendu de $\mathcal{C}$ est tel que suit :




– Lorsque $\mathcal{C}$ visite le nœud $@$, il *devine* un *environnement*, qui décrit un ensemble d'informations telles que "l'évaluation du $i$-ème argument de cette application retourne au $j$-ème fils de $\phi_i$, dans l'état $q_1$, et en ayant rencontré au cours du saut la

priorité minimale $m_1$, ou alors elle peut revenir à ce $j$-ème fils dans l'état $q_2$ en ayant recontré au cours du saut la priorité minimale $m_2$".

– Sur les branches décrivant des arguments, l'automate vérifie que l'environnement qu'il a deviné précédemment ne contient ni prédiction erronée, ni prédiction superflue.

– Sur la branche qui correspond à la direction 0 depuis @, qui représente le "code" de la "fonction" auquel on applique des arguments, on propage l'environnement, et lorsqu'on rencontre une variable, on étiquette ses fils en fonction des valeurs de retour de $\beta$-réduction devinées par l'environnement - qui seront correctes sur un processus d'étiquetage qui termine.

4.1.2. *Profils de variables, profils actifs, environnements*

Afin de définir plus formellement $\mathcal{C}$, il nous faut introduire les structures d'étiquetage que l'on vient d'esquisser.

**Définition 21 (Profil de variable)** *Un profil de variable pour $\mathcal{B}$ est un quadruplet $(\phi, q, m, c)$, où $\phi$ est la variable décrite par le profil (de type $A_1 \rightarrow \cdots \rightarrow A_n \rightarrow o$), $q$ un état de $\mathcal{B}$, $m$ une priorité de $\mathcal{B}$ et $c$ un ensemble de profils de variables décrivant des variables de types $A_i$, qu'on appelle l'*interface *du profil.*

**Définition 22 (Profil actif)** *Un profil* actif *d'un automate alternant à parité $\mathcal{B}$ est un quadruplet $(\phi, q, m, c)^b$, consistant en un profil de variable $(\phi, q, m, c)$ et en un booléen $b$ valant faux à un stade de l'étiquetage si la priorité $m$ n'a pas encore été rencontrée et vrai si c'est la priorité minimale vue jusqu'ici. L'automate sera défini pour bloquer dans le cas où l'on rencontrerait une priorité inférieure à $m$ au cours de l'étiquetage.*

**Définition 23 (Environnement)** *Un environnement $\rho$ pour l'automate alternant à parité $\mathcal{B}$ sur $\lambda(G)$ est un ensemble de profils actifs décrivant les variables apparaissant dans $\lambda(G)$.*

Un *profil de variable* décrit une variable $\phi$, donnant l'état $q$ de l'automate que l'on simule (celui qui décrit la formule de $\mu$-calcul modal à vérifier) lorsque l'automate $\mathcal{C}$ recontre $\phi$, ainsi que la priorité $m$ la plus faible rencontrée entre la racine et la dernière occurence de $\phi$ rencontrée (et liée par le *même* nœud $\lambda\phi$, pas par une copie correspondant au calcul d'une autre application similaire), et donne une *interface* qui décrit les profils de variables avec lesquels on étiquettera les fils des nœuds d'étiquette $\phi$. Un *profil actif* ajoute un booléen qui permettra de contrôler la véracité de l'information sur la priorité minimale recontrée. Un environnement donne ce type d'informations pour toutes les variables de $\lambda(G)$ ; il peut y avoir plusieurs profils pour une même variable si sa réduction peut amener à plusieurs traversées différentes (donc, si on rencontre au cours de sa réduction un symbole de $\Sigma$ d'arité supérieure ou égale à 2).

4.1.3. *Description du comportement de $\mathcal{C}$*

Les arbres d'exécution de $\mathcal{C}$ sur $\lambda(G)$ seront annotés par l'automate d'un état (de $\mathcal{C}$) de forme $q\ \rho$ ou $q\ \rho\ \theta$, avec $q$ un état de $\mathcal{B}$, $\rho$ un environnement pour $\mathcal{B}$ sur $\lambda(G)$ et $\theta$ un profil de variable que l'on dira *distingué*. L'état initial est $q_0\ \varnothing$ et la priorité d'un état de $\mathcal{C}$ (de forme $q\ \rho$ ou $q\ \rho\ \theta$, donc) sera :

– Si l'état est de forme $q\ \rho$, celle de $q$ dans $\mathcal{B}$

– Si l'état est de forme $q\ \rho\ \theta$, celle contenue dans $\theta$.

On se contentera de décrire le comportement de la fonction de transition de $\mathcal{C}$ :

**Sur un nœud @, annoté par l'état $q\ \rho$ lors de l'exécution de $\mathcal{C}$ :** Dans cette situation, $\mathcal{C}$ doit étiqueter les $n+1$ fils de @, sachant que cette structure encode l'application des $n$ fils les plus à droite au fils le plus à gauche. Ce fils de direction 0 est étiqueté $\lambda\phi_1 \cdots \phi_n$, et chaque $\phi_i$ décrit le $i$-ème argument de cette application. $\mathcal{C}$ procède comme suit :

– Il commence par deviner un ensemble $c = \{\theta_j = (\xi_{i_j}, q_{l_j}, m_j, c_j)\}$ de profils actifs décrivant les $\phi_i$ - il peut y en avoir plusieurs pour une même variable, si sa $\beta$-évaluation peut avoir plusieurs résultats différents[21].

– Ensuite, il étiquette la direction 0 avec l'état $q\ c$ : on est toujours en train de simuler l'état $q$ de $\mathcal{B}$ dans cette direction, et $c$ donne, si tout va bien, les résultats de toutes les $\beta$-réductions, donc de tous les sauts effectués par les traversées, qu'un automate d'arbre ne peut pas simuler.

– Dans les $n$ autres directions : $\rho$ donne des informations sur des variables correspondant à des applications précédentes ; pour chacune de ces directions, l'automate devine quelles variables de $\rho$ seront nécessaires au calcul (tout profil dans $\rho$ doit être utilisé par la suite dans au moins une direction, sinon l'automate bloque), et étiquette le fils de direction $i_j$ avec l'état $q_{l_j}$ de $\mathcal{B}$, le profil distingué $\theta_j$, et l'environnement donné par l'union :

- des profils correspondant aux variables de $\rho$ dont l'utilité dans cette direction a été prédite, en prenant soin de mettre la valeur $vrai$ dans les booléens des profils actifs si la priorité minimale qu'ils décrivent est $m_j$ ($\mathcal{C}$ est supposé bloquer ici s'il trouve une priorité minimale dans l'un des profils actifs qui est plus grande que la priorité de $q_{l_j}$)

- et des $c_j$ dont on met à jour le booléen, cette fois en comparant la priorité minimale qu'ils décrivent à $\Omega(q_{l_j})$.

**Sur un nœud $\lambda$, annoté $q\ \rho$ ou $q\ \rho\ \theta$ :** Un tel nœud n'a qu'un fils, $\mathcal{C}$ l'annote avec l'état $q\ \rho\ \tau$ (où $\tau^b \in \rho$ est un profil actif d'une variable $\psi$) s'il devine que

21. Rappelons qu'il n'y a pas ici de réel non-déterminisme, simplement un choix fait par l'environnement de la direction à calculer (pour le programme) lorsqu'un symbole de $\Sigma$ ayant une arité au moins 2 est rencontré.

l'étiquette de ce fils est une variable $\psi$ ; s'il devine que le fils n'est pas une variable, il l'annote $q\ \rho$.

**Sur les nœuds correspondant à un élément de $\Sigma$ nodes, annotés** $q\ \rho$ **:** $\mathcal{C}$ doit tout d'abord deviner un ensemble d'états et de directions $\{(i_1, q_{j_1}), \cdots, (i_k, q_{j_k})\}$ vérifiant $\delta(q, f)$ où $\delta$ est la fonction de transition de $\mathcal{B}$ : n'oublions pas que l'on cherche conjointement à simuler les traversées et le comportement de l'automate testant la propriété logique que l'on veut vérifier. $\mathcal{C}$ devine ensuite quels profils actifs de $\rho$ serviront dans une direction donnée $i_l$ et étiquette le fils en direction $i_l$ avec l'état $q_{j_l}$ et l'environnement obtenu comme union des profils actifs de $\rho$ dont l'automate $\mathcal{C}$ prédit l'utilité dans cette direction, dont on met à jour les booléens, de sorte qu'ils valent maintenant $vrai$ si la priorité minimale qu'ils décrivent est égale à $\Omega(q_{j_l})$ - et le processus bloque si elle est strictement supérieure. Chaque profil actif de $\rho$ doit être propagé dans au moins une direction, le but étant de forcer $\mathcal{C}$ à toujours deviner l'environnement le plus petit possible.

**Sur les nœuds correspondant à des variable, annotés** $q\ \rho\ \theta$ **:** Sur un tel nœud $\phi$, le nœud $\lambda$ qui précède immédiatement est censé avoir distingué un profil $\theta = (\phi, q, m, c)$ pour $\phi$, tel que $\theta^t \in \rho$ (si ces conditions ne sont pas remplies, $\mathcal{C}$ bloque). Ce profil distingué décrit le résultat de l'évaluation correspondant au saut de $\beta$-réduction effectué par une traversée, et $c = \{\theta_j = (\xi_{i_j}, q_{l_j}, m_j, c_j)\}$ décrit les profils actifs à utiliser pour étiqueter les fils de $\phi$ (ainsi, $c = \emptyset$ si et seulement si $\phi$ est d'ordre 0) - il n'y a rien de plus à vérifier ici, tout est fait dans les autres directions. $\mathcal{C}$ procède alors comme suit :

– Pour commencer, il devine s'il rencontrera à nouveau $\phi$ dans la portée de son lieur actuel : on pose $\rho' = \rho$ s'il prédit que ce sera le cas, et $\rho' = \rho \setminus \theta$ sinon - n'oublions pas qu'un environnement contenant des données superflues est un environnement qui amènera à un blocage au cours de l'exécution de $\mathcal{C}$.

– Ensuite, pour chaque direction $i_j$, l'automate devine quels profils de $\rho'$ y seront utiles, chaque profil devant là encore être utile à au moins une direction sous peine de blocage, et étiquette le fils de direction $i_j$ avec l'état $q_{l_j}$, le profil distingué $\theta_j$, et l'environnement formé de l'union :

- des profils actifs dont l'utilité dans cette direction a été prédite, dont on met à jour le booléen à la valeur $vrai$ si la priorité minimale décrite par le profil est $m_j$ (comme auparavant, on bloque si elle est strictement supérieure),

- et des profils actifs de $c_j$, dont on met à jour le booléen à la valeur $vrai$ si la priorité minimale décrite par le profil correspondant est $\Omega(q_{l_j})$ (là encore, il y a blocage si la priorité minimale contenue dans le profil est fausse).

Regardons ce que cela donne sur un exemple :

**Exemple 10 (Un arbre d'exécution de $\mathcal{C}$)** *On considère l'automate simulateur de traversées $\mathcal{C}$ pour l'automate alternant à parité $\mathcal{B}$ d'états $1$ et $2$, de priorité leur étiquette, d'état initial $1$, et $\delta$ donnée par :*

- $\delta(1, g) = ((1, 1) \wedge (2, 1)) \vee ((1, 2) \wedge (2, 1))$,
- $\delta(1, a) = \delta(2, a) = \top$,
- $\bot$ *partout ailleurs,*

*qu'on exécute sur l'arbre $\lambda(G)$ de l'exemple 4. On note $\theta = (\phi, 1, 1, \{\theta_0\})$, $\theta_0 = (y, 1, 1, \emptyset)$, $\theta_1 = (z, 1, 1, \emptyset)$, $\theta_2 = (z, 0, 0, \emptyset)$. L'arbre de la figure 10 est alors un arbre d'exécution de cet automate alternant sur $\lambda(G)$ (on a ajouté des identifiants aux nœuds par commodité de lecture).*

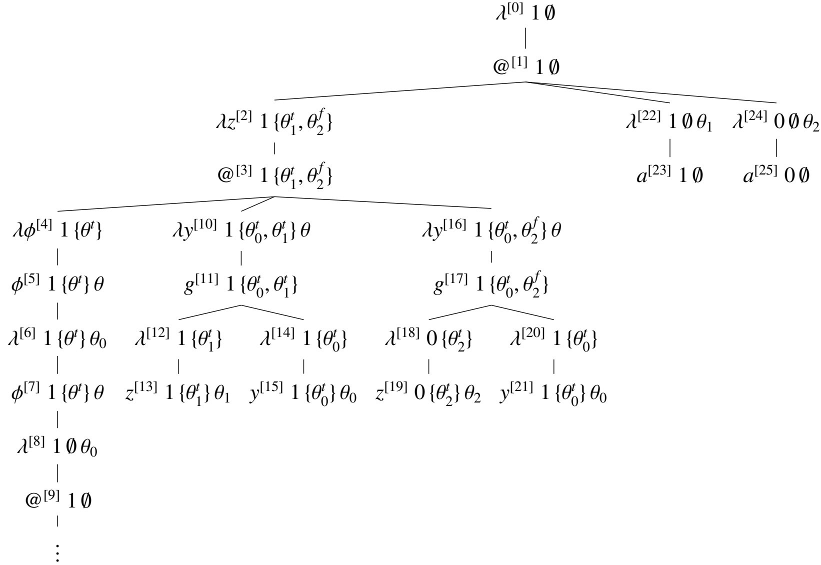



**Figure 3.** *Arbre d'exécution d'un automate alternant simulateur de traversées.*

– *Pour commencer, $\mathcal{C}$ étiquette la racine $[0]$ avec son état initial $1$ $\emptyset$. Ensuite, comme la racine est un nœud $\lambda$, dont $\mathcal{C}$ prédit que le fils $[1]$ n'est pas une variable, il étiquette ce fils $1$ $\emptyset$.*

– *$[1]$ est un nœud $@$, marquant le début d'une application. $\mathcal{C}$ devine que $z$ va être rencontré, et qu'il faut donc deviner ce que donnerait l'évaluation de l'argument qu'il dénote : $\mathcal{C}$ prédit que cette évaluation correspond aux états de retour de $\mathcal{B}$ $1$ ou $0$, en ayant respectivement rencontré au cours de l'évaluation les plus faibles priorités $1$ et $0$ ; il devine aussi que dans l'exécution courante les deux cas arriveront, et prend donc $c = \{\theta_1^t, \theta_2^f\}$. Chacune de ces deux valeurs va être vérifiée dans l'arbre d'exécution que $\mathcal{C}$ construit, respectivement en direction $1$ et en direction $2$, avec un environnement*

*vide (puisque $\rho$ et les interfaces de $\theta_1$ et $\theta_2$ le sont), en distinguant le profil $\theta_1$ en direction* $1$ *et $\theta_2$ en direction* $2$*.*

*– Dans ces deux directions* $1$ *et* $2$*, $\mathcal{C}$ devine sur les nœuds $\lambda$ (*$[22]$ *et* $[24]$*) que leur fils $a$ n'est pas une variable, et l'étiquette donc* $1\ \emptyset$ *; ensuite sur $a$ (nœuds* $[23]$ *et* $[25]$*) l'automate arrête l'exploration mais sans bloquer puisque* $\delta(1, a) = \delta(2, a) = \top$*.*

*– Dans la direction* $0$*, l'automate rencontre d'abord le nœud* $[2]$*, il contient un $\lambda$, et l'automate prédit que son fils n'est pas une variable et ne distingue donc pas l'un des profils de variables de $\rho$.*

*– En* $[3]$*, $\mathcal{C}$ commence par deviner un profil actif $\theta$ pour $\phi$, qui est la nouvelle variable introduite dans le fils de direction* $0$*, et qui désigne le seul argument de cette nouvelle application. $\theta$ contient dans son interface l'information sur l'évaluation de l'argument correspondant à $\phi$ : selon $\theta$ elle amène (dans le contexte correspondant à la construction de cet arbre d'exécution) à l'étiquetage du fils de $\phi$ par le profil $\theta_0$ ; pour vérifier cette prédiction et propager celles concernant $z$, $\mathcal{C}$ affecte les environnements* $\{\theta_0^t, \theta_1^t\}$ *et* $\{\theta_0^t, \theta_2^f\}$ *aux fils de directions respectives* $1$ *et* $2$ *- la vérification simultanée étant là encore permise par la structure d'automate alternant de $\mathcal{C}$.*

*– En direction* $1$*, sur le nœud $\lambda$ situé en* $[10]$*, l'automate devine que le fils n'est pas une variable et ne lui attribue donc pas un profil distingué. Ensuite, l'automate arrive en $g$ et répartit les profils contenus dans l'environnement entre les deux fils (il pourrait en avoir affecté certains plusieurs fois, il faut simplement tous les propager dans au moins une direction), puis sur les nœuds $\lambda$ il devine que les fils sont des variables, et distingue donc un profil qui leur convient et avec lesquels il les étiquette. L'automate visite ensuite ces nœuds, y vérifie que le profil distingué est cohérent, et s'arrête donc sans bloquer. En direction* $2$*, un processus similaire a lieu.*

*– Dans la direction* $0$*, $\mathcal{C}$ arrive au nœud* $[4]$*, qui contient un $\lambda$. Il prédit que son fils sera $\phi$ et distingue un profil pour celui-ci - il n'y en a qu'un qui corresponde à $\phi$, c'est donc $\theta$ qui sera distingué.*

*– $\mathcal{C}$ arrive en* $[5]$ *: c'est une variable dont l'ordre n'est pas nul, avec un unique fils. $\mathcal{C}$ devine que $\phi$ sera rencontrée à nouveau dans la portée de son lieur (qui est le nœud* $[4]$*) et laisse donc son profil dans l'environnement. $\theta_0$ est affecté comme profil distingué au nœud* $[6]$*, et l'état reste* $1$*, puisque $\theta_0$ indique que l'évaluation de l'argument à substituer à $\phi$ retourne dans l'état* $1$*.*

*– Au nœud* $[6]$*, on a encore un $\lambda$ : $\mathcal{C}$ devine que son fils est $\phi$ et lui donne donc un profil distingué convenable.*

*–* $[7]$ *requiert à nouveau la simulation d'un saut effectué dans l'arbre par la traversée, il suffit donc ici de suivre l'information sur celui-ci que contient $\theta_0$. $\mathcal{C}$ devine que $\phi$ ne sera plus réutilisé dans la portée de son lieur, et le supprime donc de l'environnement qu'il affecte à* $[8]$ *; $\theta_0$ est ajouté comme profil distingué à* $[8]$ *comme le préconise la simulation du saut de la traversée.*

*– Sur le nœud $\lambda$* $[8]$*, $\mathcal{C}$ devine que le fils n'est pas une variable et ne lui distingue donc pas de profil.*

– *Au nœud* $[9]$ *commence une nouvelle application, ainsi qu'un processus similaire.*

L'automate $\mathcal{C}$ dont on a décrit le comportement général simule les arbres de traversées de $\mathcal{B}$, c'est à dire qu'il y a un arbre d'exécution acceptant de $\mathcal{C}$ sur $\lambda(G)$ si et seulement s'il y a un arbre de traversées acceptant de $\mathcal{B}$ sur $\lambda(G)$.[22]

## 4.2. *Une machine équivalente aux schémas de récursion d'ordre supérieur*

Une autre façon de simuler les traversées de $\lambda(G)$ fut introduite dans (Hague *et al.*, 2008). Elle utilise des automates à pile d'un genre particulier, appelés *automates à pile effondrants (d'ordre $n$)*, abrégé en $n$-CPDA[23], et dont l'expressivité est équivalente à celle des schémas de récursion (d'ordre $n$). De plus, le comportement de ces automates est particulièrement adapté à la simulation des traversées sur un arbre $\lambda(G)$. Après avoir introduit ce nouveau formalisme, nous verrons quel lien unit les traversées à ces automates.

### 4.2.1. *Automates à pile effondrants d'ordre supérieur*

Les $n$-CPDA sont informellement des automates à pile, tels que les éléments de cette pile peuvent eux-mêmes être des piles de piles de piles . . . munies de pointeurs vers des piles les précédant dans l'empilement, et d'une opération d'effondrement enlevant d'un coup tout ce qui est situé entre l'élément au sommet de la pile et la cible de son pointeur. Ces automates génèrent des arbres.

**Définition 24 ($n$-piles à pointeurs)** *Une* $0$*-pile est juste un symbole de l'alphabet de pile ; une* $n$*-pile est une suite de* $(n-1)$*-piles, qui sont des suites de* $(n-2)$*-piles . . . On demande de plus que dans une* $1$*-pile apparaissant dans cette* $n$*-pile le symbole le plus en profondeur soit toujours le symbole distingué* $\perp$*, tous les autres symboles devant être différents de ce marqueur de fond de pile et étant munis d'un pointeur vers une pile (d'ordre quelconque) apparaissant en-dessous d'eux dans la* $n$*-pile.*

**Définition 25 (Opérations sur une $n$-pile)** *Sur une* $n$*-pile, on se munit des opérations suivantes, dont l'ensemble est noté* $Op_n$ *:*

– $top_i$ *renvoie la* $(i-1)$*-pile du dessus,*

– $pop_i$ *enlève la* $(i-1)$*-pile du dessus,*

– *push à l'ordre 1 :* $push_1^{a,k}$ *ajoute* $a$ *au-dessus de la* $1$*-pile du dessus, avec un pointeur vers la* $(k-1)$*-pile la plus proche,*

– $push_i$ *à l'ordre* $i \geq 2$ *duplique la* $(i-1)$*-pile du dessus ; cette opération préserve la structure des liens : les liens pointant hors de la* $(i-1)$*-pile que l'on duplique*

22. La preuve se trouve dans (Ong, 2006, Sections 4 et 5).
23. En anglais, on parle de *Collapsible PushDown Automata*.

*pointent vers les mêmes cibles dans la pile dupliquée, et les liens pointant à l'intérieur de la pile originale pointent, dans la pile dupliquée, vers leurs copies dans cette* $(i-1)$*-pile dupliquée.*

*–* $collapse$ *enlève tout ce qui se trouve entre le symbole au sommet de la pile et la cible de son pointeur (qui est conservée).*

**Exemple 11 (Opérations sur une** $3$**-pile)** *Soit la* $3$*-pile* $s = [[[\bot a]][[\bot][\bot a]]]$*. A partir de maintenant, on ne représentera pas les pointeurs d'un symbole vers celui le précédant immédiatement (au sein d'une même* $1$*-pile, donc). Considérons quelques opérations sur* $s$ *:*

– $push_1^{b,2}\, s \;=\; s_1 \;=\; [[[\bot\, a]]\,[[\bot]\,[\bot\, a\, b]]]$

– $push_1^{c,3}\, s_1 \;=\; s_2 \;=\; [[[\bot\, a]]\,[[\bot]\,[\bot\, a\, b\, c]]]$

– $push_2\, s_2 \;=\; s_3 \;=\; [[[\bot\, a]]\,[[\bot]\,[\bot\, a\, b\, c]\,[\bot\, a\, b\, c]]]$

– $push_3\, s_2 \;=\; s_3' \;=\; [[[\bot\, a]]\,[[\bot]\,[\bot\, a\, b\, c]]\,[[\bot]\,[\bot\, a\, b\, c]]]$

– $collapse\, s_3 \;=\; collapse\, s_3' \;=\; [[[\bot\, a]]]$

*Remarquons la différence de structure de pointeurs entre* $s_3$ *et* $s_3'$ *: dans la sous-pile que l'on duplique, les liens pointant en-dehors de celle-ci sont, dans la pile dupliquée, de même cible que dans l'originale, alors que les liens pointant à l'intérieur de la pile originale sont, dans la pile dupliquée, de cible dans la pile dupliquée.*

**Définition 26 (**$n$**-CPDA)** *Un* $n$*-CPDA est un quintuplet* $\mathcal{A} = <\Sigma, \Gamma, Q, \delta, q_0>$*, où* $\Sigma$ *est un alphabet de sortie à arité,* $\Gamma$ *est l'alphabet de pile (avec un symbole distingué de fond de pile* $\bot$*),* $Q$ *est un ensemble d'états fini,* $q_0$ *est l'état initial et* $\delta : Q \times \Gamma \to Q \times Op_n \cup \{(f; q_1 \cdots q_{ar(f)}) / f \in \Sigma, q_i \in Q\}$ *est la fonction de transition, envoyant un couple formé d'un état et d'un symbole de pile vers, selon le cas, un nouvel état et une opération de pile, ou vers une opération de sortie (*$\delta$ *ne peut pas ajouter ou retirer de symbole* $\bot$*).*
*On définit plus précisément les transitions sur des* configurations $(q, s)$ *avec* $q \in Q$ *et* $s$ *une* $n$*-pile ; la configuration initiale est* $(q_0, \bot_n)$ *où* $\bot_n$ *est la* $n$*-pile vide* $[\cdots[\bot]\cdots]$*.*
*Il y a trois sortes de transitions :*

*– Coups internes :* $(q, s) \xrightarrow{(q',\theta)} (q', s')$ *si* $\delta(q, top_1\, s) = (q', \theta)$ *et* $s' = \theta(s)$

*– Coups du programme :* $(q, s) \xrightarrow{(f, q_1, \cdots, q_{ar(f)})} (f; q_1, \cdots, q_{ar(f)}; s)$ *si* $\delta(q, top_1\, s) = (f, q_1, \cdots, q_{ar(f)})$

– *Coups de l'environnement :* $(f, q_1, \cdots, q_{ar(f)}; s) \xrightarrow{(f,i)} (q_i, s)$ *pour tout* $1 \leq i \leq ar(f)$.

Un *chemin d'exécution* est une suite de configurations reliées par des transitions telles que ci-dessus, et commençant sur la configuration initiale.
On retrouve dans cette description le vocabulaire de la sémantique des jeux : l'idée est que les coups internes correspondent à des opérations usuellement cachées, que l'on considère ici puisque l'on fait appel à une *sémantique des jeux révélée*, tandis que les coups de sortie correspondent à ceux du programme et de l'environnement et sont ceux obtenus dans le cadre classique où l'interaction est effacée lors de la composition des stratégies. L'idée est qu'un coup du programme correspond à une valeur de retour : "J'ai calculé le prochain symbole sur l'arbre produit par le schéma $G$, c'est $f \in \Sigma$", et l'environnement pose une nouvelle question : "Bien, je sais que $f$ est d'arité $n$, donne-moi le symbole suivant sur la $(1 \leq i \leq n)$-ème branche partant de $f$". Les machines que nous considèrerons dans la suite seront *déterministes* bien qu'ayant plusieurs chemins d'exécution possibles : le seul choix ayant une influence sur le calcul aura lieu lors des coups de l'environnement, et pour un choix fixé de ces coups il n'y aura qu'un unique chemin d'exécution. Ceci correspond à la structure des traversées : hormis après la visite d'un symbole de $\Sigma$ où l'environnement demande le prochain symbole de $[\![G]\!]$ dans une direction donnée, tout est déterministe.

4.2.2. *Les $n$-CPDA simulent les traversées*

Les $n$-CPDA et les schémas de récursion d'ordre $n$ sont équi-expressifs : ils génèrent les mêmes arbres. L'idée[24] que l'on peut encoder les $n$-piles comme des non-terminaux bien typés d'un schéma de récursion dont les règles encodent la dynamique d'un $n$-CPDA donné, et qu'inversement on peut simuler les traversées sur un arbre de calcul de schéma de récursion $\lambda(G)$ à l'aide d'un $n$-CPDA simulateur de traversées ; la correspondance chemins-traversées impliquant alors que le $n$-CPDA calcule toute branche de $[\![G]\!]$ demandée par l'environnement, et donc qu'il calcule $[\![G]\!]$. Nous esquisserons simplement ici cette inclusion inverse, en insistant surtout sur la structure du $n$-CPDA simulateur de traversées.
On notera $E_i(u)$ le fils en direction $i$ du nœud $u$ de $\lambda(G)$ et $v_0$ la racine de $\lambda(G)$.

**Définition 27 (Le $n$-CPDA simulateur de traversées $\mathcal{A}$)** *Etant donné un schéma de récursion $G$ d'ordre $n$, on définit son $n$-CPDA* simulateur de traversées *comme étant l'automate dont l'alphabet de sortie est celui de $G$, dont l'alphabet de pile est celui donné par l'ensemble des étiquettes des nœuds de $\lambda(G)$, de configuration initiale* $(q_0, [\cdots[\bot v_0]\cdots])$[25] *et dont l'action de la fonction de transition est donnée selon le*

24. Les preuves se trouvent dans (Hague *et al.*, 2008).
25. Ceci ne satisfait pas exactement notre définition précédente, mais il devrait être évident qu'il suffit pour s'y ramener d'ajouter un état supplémentaire, de le rendre initial, et de définir la fonction de transition comme empilant $v_0$ et allant en $q_0$ depuis cet état pour ne plus jamais y revenir.

*symbole de dessus de pile* $u$[26] *:*

*– Si* $u$ *est une étiquette* @, $\delta(u) = push_1^{E_0(u)}$.

*– Si* $u$ *est une étiquette* $f \in \Sigma$, $\delta(u) = push_1^{E_i(u)}$, *où* $1 \leq i \leq ar(f)$ *est la direction choisie par l'environnement*[27].

*– Si* $u$ *est une étiquette* $\lambda$, $\delta(u) = push_1^{E_1(u)}$.

*– Si* $u$ *est une étiquette* $\phi$ *avec* $\phi$ *une variable d'ordre* $l$*, étant le* $i$*-ème paramètre de son lieur, que son lieur est le fils de direction* $j$ *de son nœud parent*[28]*, et qu'il y a* $p$ *nœuds sur le chemin reliant* $\phi$ *à son lieur dans* $\lambda(G)$ *(*$\phi$ *et son lieur compris), on a :*

*- Si* $l \geq 1$ *et* $j = 0$, $\delta(u) = push_{n-l+1}\,;\, pop_1^p\,;\, push_1^{E_i(top_1),n-l+1}$ *(la composition séquentielle d'opérations sur la* $n$*-pile étant notée* ; *et la puissance réfère à l'itération de cette notion de composition)*

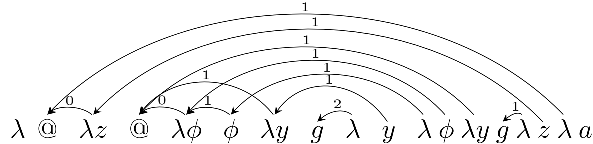
*- Si* $l, j \geq 1$, $\delta(u) = push_{n-l+1}\,;\, pop_1^{p-1}\,;\, collapse\,;\, push_1^{E_i(top_1),n-l+1}$

*- Si* $l = 0$ *et* $j = 0$, $\delta(u) = pop_1^p\,;\, push_1^{E_i(top_1)}$

*- Si* $l = 0$ *et* $j \geq 1$, $\delta(u) = pop_1^{p-1}\,;\, collapse\,;\, push_1^{E_i(top_1)}$

Regardons comment ceci se comporte sur un exemple :

**Exemple 12 (Traversées et chemins d'exécutions)** *Considérons le* 2*-CPDA simulateur de traversées pour le schéma* $G$ *de l'exemple 4. Nous avons vu dans l'exemple 5 que la suite justifiée suivante est une traversée sur* $\lambda(G)$ *:*




*Représentons les configurations de pile correspondant au calcul de cette traversée par le* 2*-CPDA simulateur de traversées associé à* $G$*. On ne représentera pas les*

26. Ici encore, on s'éloigne un peu de la définition formelle des $n$-CPDA pour plus de simplicité : on se permet la composition de plusieurs opérations de piles, ce qui nécessiterait formellement d'introduire des états intermédiaires pour les réaliser tour à tour. On ne considèrera donc pas formellement les états de $\mathcal{A}$, cela n'apportant rien à la compréhension générale de son action.
27. Cette transition est celle correspondant à la succession d'un coup du programme (répondant à la dernière question de l'environnement en donnant le symbole de $[\![G]\!]$ calculé) et à une nouvelle question de l'environnement, correspondant au choix de la direction à calculer. C'est, dans cette énumération, la seule situation correspondant à des coups non-internes.
28. Si $j = 0$, on est dans la situation où l'on lit le terme dans lequel il faudra faire des substitutions ; moralement, c'est le "code" d'une fonction. Si $j \geq 1$, on est en train d'explorer un argument, donc en train de faire une $\beta$-réduction.

*étapes triviales (l'utilisation de $\rightarrow^*$ désignera donc l'itération d'étapes de calcul de l'automate sur la pile) ni les pointeurs triviaux (vers le symbole immédiatement précédent).*

$$
\begin{array}{lll}
(1) & & [\,[\,\bot\ \lambda\,]\,] \\
(2) & \rightarrow^* & [\,[\,\bot\ \lambda\ @\ \lambda z\ @\ \lambda\phi\ \phi\,]\,] \\
(3) & \rightarrow & [\,[\,\bot\ \lambda\ @\ \lambda z\ @\ \lambda\phi\ \phi\,]\,[\,\bot\ \lambda\ @\ \lambda z\ @\ \lambda y\,]\,] \\
(4) & \rightarrow^* & [\,[\,\bot\ \lambda\ @\ \lambda z\ @\ \lambda\phi\ \phi\,]\,[\,\bot\ \lambda\ @\ \lambda z\ @\ \lambda y\ g\ \lambda\ y\,]\,] \\
(5) & \rightarrow & [\,[\,\bot\ \lambda\ @\ \lambda z\ @\ \lambda\phi\ \phi\ \lambda\,]\,] \\
(6) & \rightarrow & [\,[\,\bot\ \lambda\ @\ \lambda z\ @\ \lambda\phi\ \phi\ \lambda\ \phi\,]\,] \\
(7) & \rightarrow & [\,[\,\bot\ \lambda\ @\ \lambda z\ @\ \lambda\phi\ \phi\ \lambda\ \phi\,]\,[\,\bot\ \lambda\ @\ \lambda z\ @\ \lambda y\,]\,] \\
(8) & \rightarrow^* & [\,[\,\bot\ \lambda\ @\ \lambda z\ @\ \lambda\phi\ \phi\ \lambda\ \phi\,]\,[\,\bot\ \lambda\ @\ \lambda z\ @\ \lambda y\ g\ \lambda\ z\,]\,] \\
(9) & \rightarrow & [\,[\,\bot\ \lambda\ @\ \lambda z\ @\ \lambda\phi\ \phi\ \lambda\ \phi\,]\,[\,\bot\ \lambda\ @\ \lambda\,]\,] \\
(10) & \rightarrow & [\,[\,\bot\ \lambda\ @\ \lambda z\ @\ \lambda\phi\ \phi\ \lambda\ \phi\,]\,[\,\bot\ \lambda\ @\ \lambda\ a\,]\,]
\end{array}
$$

*– De l'étape $1$ à l'étape $2$ : l'automate simule une descente dans $\lambda(G)$ commençant à sa racine, prenant la direction $0$ lorsqu'il recontre un $@$, et stockant dans sa pile ce qu'il lit.*

*– De l'étape $2$ à l'étape $3$ : le sommet de la pile est une variable d'ordre $l = 1$ ; on a $j = 0$ puisque $\lambda\phi$ est le fils de direction $0$ de son père $@$. On applique donc la première règle pour les variables : l'automate duplique la $1$-pile du dessus à l'aide d'une opération $push_2$, puis enlève tout symbole de cette nouvelle pile jusqu'à ce qu'il trouve le $@$ correspondant à l'application dont $\phi$ est un des arguments, ensuite il met sur cette nouvelle pile le fils de direction $1$ de $@$ (correspondant donc à l'argument décrit par $\phi$) avec un lien vers la $1$-pile qui vient d'être dupliquée, de sorte que l'effondrement depuis cette configuration de pile ramène à la situation d'avant le début de la simulation du saut de $\beta$-réduction dans $\lambda(G)$.*

*– De l'étape $3$ à l'étape $4$ : l'automate explore le code de l'argument ; lors de la visite de $g$, l'environnement demande le calcul de la direction $2$.*

*– De l'étape $4$ à l'étape $5$ : le sommet de la pile est une variable d'ordre $0$, on a donc fini l'évaluation de l'argument désigné par $\phi$. On a trouvé $y$, qui représente l'unique fils de $\phi$, qu'il faut donc maintenant visiter. Pour $y$, $j = 1$ : l'automate enlève donc un par un tous les symboles du sommet de la pile jusqu'à arriver au lieur $\lambda y$ de $y$, lance une opération d'effondrement qui ramène la pile à sa situation d'avant la simulation de la réduction de l'argument, et empile le fils de $\phi$ pour y continuer le calcul.*

*– De l'étape $5$ à l'étape $6$ : l'automate continue la descente dans l'arbre jusqu'à ce qu'il trouve $\phi$.*

*– De l'étape $6$ à l'étape $7$ : le comportement est similaire à celui menant de l'étape $2$ à l'étape $3$.*

*– De l'étape $7$ à l'étape $8$ : l'automate explore le code de l'argument ; lors de la visite de $g$, l'environnement demande le calcul de la direction $1$.*

*– De l'étape $8$ à l'étape $9$ : le sommet de la pile est $z$, qui est une variable d'ordre $0$ : le calcul continue sur l'argument désigné par $z$. On est ici dans le cas de la troisième règle pour les variables (le lieur de $z$ étant un fils de direction $0$), l'automate dépile donc les symboles jusqu'à arriver au $@$ qui est le père de $\lambda z$, puis descend dans sa première direction, qui est celle désignée par $z$, et empile le symbole correspondant.*

*– De l'étape $9$ à l'étape $10$ : l'automate descend dans $\lambda(G)$ et y trouve $a$. Ce symbole étant d'arité $0$, l'environnement n'a pas de coup possible et le calcul s'arrête. Nous reconnaissons dans ce calcul celui de la branche $g \cdot g \cdot a$ de $[\![G]\!]$.*

Dans cet exemple on n'a pas utilisé la deuxième règle pour les variables : elle n'intervient qu'à partir de l'ordre 3. On pourra trouver un exemple de traversée à cet ordre dans (Grellois, 2010, Annexe A). A partir de cet ordre, il peut falloir simuler des $\beta$-réductions au cours de telles réductions, ce qui amène à faire des sauts dans $\lambda(G)$ pendant de tels sauts : l'utilisation de $push_{n-l+1}$ permet alors de sauvegarder le contexte local, pour pouvoir en modifier une copie et simuler ainsi un nouveau saut sans perdre l'information permettant de revenir à l'origine de celui-ci.

4.2.3. *Traversées et configurations de $n$-piles*

Les 2-piles vues à l'exemple précédent ne donnent pas exactement les traversées sur $\lambda(G)$, il faut un peu de mise en forme sur les $n$-piles et sur les traversées pour arriver à la propriété générale de *bisimulation* que l'on devine :

– Sur les $n$-piles : si $s$ est une $n$-pile, on définit la suite $\bar{s}$ obtenue en lisant $s$ en partant "de la droite" (donc de son sommet), en ignorant tout symbole [ , ] , ou $\perp$, et en suivant les pointeurs ($\bar{s}$ ne contient donc pas les symboles strictement contenus entre la source et le but d'un pointeur qu'elle suit). Si l'on reprend l'exemple 12, on a pour l'étape 8, $\bar{s} = @ \, \lambda z \, @ \, \lambda\phi \, \phi \, \lambda \, \phi \, \lambda y \, g \, \lambda \, z$.

– Sur les traversées : si $t$ en est une, on définit $\hat{t}$ obtenue de $t$ en enlevant toute sous-suite justifiée $w$ contenue entre deux symboles de la forme $\overset{i}{\$ \;\; \lambda}$, où $i \geq 1$ et où \$ est une variable d'ordre $1$ ou un $@$ (ce qui implique que $w = \overset{i}{\lambda\bar{\phi} \cdots \phi_i}$ avec $\phi_i$ d'ordre 0). L'information contenue dans $\hat{t}$ suffit à reconstituer la traversée représentée : cacher les coups internes correspondant à une $\beta$-réduction à l'ordre $1$ ne change rien, puisque $\hat{t}$ contient son résultat, c'est-à-dire le nœud de retour du saut d'ordre 1 dans $\lambda(G)$.

On définit comme suit le calcul d'une traversée par une $n$-pile (d'un $n$-CPDA) :

**Définition 28** *Si $s$ est une $n$-pile accessible du $n$-CPDA simulateur de traversées de*

*$G$ et que $t$ est une traversée sur $\lambda(G)$, on dit que $s$ calcule $t$ si et seulement si les conditions suivantes sont remplies :*

– $top_2\, s = \lceil t \rceil$

– $\bar{s} = \hat{t}$

*– Si $top_2\, s = [\perp s_1 \cdots s_n]$ et que $v_1, \cdots, v_n$ sont les occurences dans $t$ qui correspondent à $s_1, \cdots, s_n$ et contribuent à $\lceil t \rceil$, alors $pop_1^{n-i}\, s$ calcule le préfixe de $t$ finissant en $v_i$ (inclus).*

*– Si $top_2\, s = [\perp s_1 \cdots s_n]$ et que $v_1, \cdots, v_n$ sont les occurences dans $t$ qui correspondent à $s_1, \cdots, s_n$ et contribuent à $\lceil t \rceil$, alors $collapse(pop_1^{n-i}\, s)$ calcule le préfixe de $t$ finissant en $v_i$ (exclu).*

Ceci amène au théorème suivant :

**Théorème 3** *Si $s$ calcule $t$, alors $s$ et $t$ sont bisimilaires :*

*– Si $s'$ est accessible depuis $s$ dans le $n$-CPDA simulateur de traversées, alors $t$ peut être étendue en $t'$ de sorte que $t$ soit un préfixe de $t'$, que $t'$ soit une traversée sur $\lambda(G)$, et que $s'$ calcule $t'$.*

*– Si $t$ peut s'étendre en $t'$ de façon que $t$ soit préfixe de $t'$ et que $t'$ soit une traversée sur $\lambda(G)$, alors il existe une $n$-pile $s'$ accessible depuis $s$ et calculant $t'$.*

Ainsi les $n$-CPDA peuvent simuler des traversées, comme les automates alternants de la section précédente. Les deux dispositifs vont donc nous permettre de répondre au problème de la vérification des schémas de récursion d'ordre supérieur à l'aide des traversées.

## 5. Vérification par résolution de jeux de parité

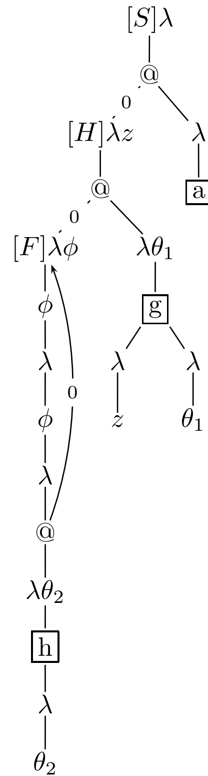
Il est temps d'utiliser les dispositifs simulant les traversées que nous avons obtenus pour résoudre le problème de la vérification des schémas de récursion d'ordre supérieur. Nous allons voir que la donnée d'un automate alternant à parité et d'un arbre induit un jeu de parité dont la résolution équivaut à l'existence d'un arbre d'exécution acceptant de cet automate sur cet arbre : on pourra appliquer ceci à $\mathcal{C}$ sur $\lambda(G)$ et le problème de vérification se réduira alors au problème bien connu de la résolution d'un jeu de parité. Les $n$-CPDA engendreront eux aussi des jeux de parité, cette fois sur l'ensemble de leurs configurations - puisque les configurations de pile permettent de simuler les traversées, ceci permettra d'effectuer la vérification sur ces traversées, et donc sur les branches de $[\![G]\!]$ par la correspondance chemins-traversées.

## 5.1. *Approche par automates alternants à parité*

### 5.1.1. *Automates alternants à parité et jeux de parité*

Un arbre d'exécution est acceptant s'il satisfait la condition de parité usuelle ; il suffit donc de construire, pour un automate alternant à parité $\mathcal{C}$ et un arbre $\lambda(G)$, un jeu dans lequel Eloïse n'a de stratégie gagnante que si et seulement s'il existe un arbre d'exécution acceptant de $\mathcal{C}$ sur $\lambda(G)$. L'idée est qu'Eloïse contrôle le choix de l'ensemble satisfaisant la fonction de transition $\delta$ de $\mathcal{C}$, de sorte qu'elle construit l'arbre d'exécution qu'elle pense acceptant, tandis qu'Abélard a le contrôle de la branche de cet arbre qui sera explorée au cours du jeu : il choisira donc une branche dont il pense qu'elle ne satisfait pas la condition de parité. Si Eloïse a une stratégie gagnante depuis ce qui correspond à la configuration initiale de l'automate alternant exécuté sur $\lambda(G)$, c'est qu'elle est en mesure de construire toutes les branches d'un arbre d'exécution de $\mathcal{C}$ sur $\lambda(G)$ et qu'aucune branche de celui-ci ne déroge à la condition de parité, de sorte qu'il est acceptant.
Afin d'obtenir un jeu *fini*, on considère au lieu de $\lambda(G)$ le graphe dont $\lambda(G)$ est le déploiement (voir la Section 2.1). On le notera Gr($G$).

**Exemple 13** *Reprenons une fois encore le schéma G de l'exemple 4. Celui-ci génère le graphe Gr(G) suivant :*




*où $[S]$, $[F]$ et $[H]$ indiquent les racines des règles correspondantes.*

A partir de Gr($G$) et de $\mathcal{C}$, on construit un jeu d'acceptance à parité comme suit : $Q$ étant l'ensemble des états de $\mathcal{C}$ et $\delta$ sa fonction de transition, on définit tout d'abord pour un nœud $v$ de Gr($G$) et un ensemble $P$ de couples de directions de fils de $v$ et d'états de $Q$ l'ensemble :

$$[P]_v = \{(u, q) \,/\, (i, q) \in P \text{ et } u \text{ est le fils de } v \text{ en direction } i\}$$

Ainsi, étant donné un ensemble $P$ d'étiquetages possibles de directions issues de $v$ par des états de $\mathcal{C}$, $[P]_v$ donne l'ensemble correspondant des nœuds et des états à leur affecter. On peut alors définir le jeu d'acceptance à parité de $\mathcal{C}$ sur Gr($G$) :

**Définition 29 (Jeu d'acceptance à parité de $\mathcal{C}$ sur Gr($G$))** *Le jeu d'acceptance à parité de $\mathcal{C}$ sur Gr($G$) a :*

– *Deux types de sommets :*

*- Les sommets contrôlés par Abélard sont les ensembles $[P]_v$ où $v$ est un nœud de Gr(G) et $P \subset \mathrm{Dir}(f) \times Q$.*

*- Les sommets contrôlés par Eloïse sont les paires de sommets de Gr(G) et d'états de $Q$.*

*– Deux types d'arêtes :*

*- Pour tout sommet appartenant à Abélard $[P]_v$, il y a pour chaque $(u, q) \in [P]_v$ une arête $[P]_v \to (u, q)$.*

*- Pour tout sommet appartenant à Eloïse $(v, q)$, il y a pour chaque $P \subset \mathrm{Dir}(f) \times Q$ satisfaisant $\delta(q, v)$ une arête $(v, q) \to [P]_v$.*

*– Une fonction de priorité $\Omega$ qui envoie les sommets $(v, q)$ sur la priorité de $q$ dans $\mathcal{C}$ et les sommets $[P]_v = \{(u_1, q_1), \cdots, (u_r, q_r)\}$ sur la plus grande priorité dans $\mathcal{C}$ parmi celles pour $q_1, \cdots, q_r$.*

Une partie dans ce jeu est un chemin sur ce graphe débutant en $(v_0, q_0)$, avec $v_0$ le nœud de Gr($G$) correspondant à la racine de $\lambda(G)$ et donc à l'axiome $S$ de $G$, et $q_0$ l'état initial de $\mathcal{C}$. Dans une partie, la structure de Gr($G$) fait que les coups alternent entre les deux joueurs. Lorsque c'est le tour d'Eloïse de jouer, elle choisit depuis le sommet qu'elle contrôle - qui décrit un état $q$ et un symbole $v$ de $\lambda(G)$ - un ensemble satisfiant $\delta(q, v)$, donc un état appartenant à Abélard. Celui-ci choisit alors une des directions qu'aurait l'arbre d'exécution correspondant à ce choix. Eloïse gagne si la priorité minimale infiniment rencontrée est paire - donc si Abélard n'a pas été capable de trouver une branche d'arbre d'exécution violant la condition de parité de l'automate alternant, et donc le comportement attendu des récursions en $\mu$-calcul modal - ou si la partie est finie et qu'elle s'achève sur une impossibilité pour Abélard de choisir une direction. Ceci ne devrait pas arriver, sauf dans le cas d'un arbre fini dans lequel la condition de parité n'a de toute façon pas de sens : Abélard veut gagner, et s'arrangera donc toujours pour aller explorer une branche infinie de $\lambda(G)$ s'il en existe une. Il devrait être clair que l'existence d'une stratégie gagnante pour Eloïse depuis $(v_0, q_0)$ implique l'existence d'un arbre d'exécution acceptant pour $\mathcal{C}$ sur $\lambda(G)$.

#### 5.1.2. *Complexité de la vérification*

Maintenant que nous avons obtenu un jeu de vérification à parité, nous allons utiliser le théorème suivant, extrait de (Jurdzinski, 2000) :

**Théorème 4 (Jurdsziński 2000)** *La région gagnante d'Eloïse et sa stratégie gagnante dans un jeu de parité à $n$ sommets, $m$ arêtes et $p \geq 2$ priorités est calculable en temps :*

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

Ceci entraîne :

**Théorème 5 (Ong 2006)** *Le problème de la vérification de la véracité d'une formule de $\mu$-calcul modal à la racine d'un arbre généré par un schéma de récursion d'ordre $n$ est $n$-EXPTIME complète, pour tout $n \geq 0$.*

Ceci provient de l'équivalence entre les assertions suivantes :

– Une formule $\phi$ de $\mu$-calcul modal est vraie à la racine de $[\![G]\!]$.

– L'automate alternant à parité équivalent à $\phi$ $\mathcal{B}_\phi$ a un arbre d'exécution acceptant sur $[\![G]\!]$.

– L'automate alternant à parité équivalent à $\phi$ $\mathcal{B}_\phi$ a un arbre de traversées acceptant sur $\lambda(G)$.

– L'automate alternant à parité $\mathcal{C}$ simulateur de traversées pour $\mathcal{B}_\phi$ a un arbre d'exécution acceptant sur $\lambda(G)$.

– Eloïse a une stratégie gagnante dans le jeu d'acceptance à parité pour $\mathcal{C}$ sur $\lambda(G)$.

et d'une étude de la taille du jeu de parité, voir (Ong, 2006, Section 6.2).

### 5.2. *Approche par $n$-CPDS*

On peut également créer un jeu de vérification sur le graphe de configurations d'un $n$-CPDA simulateur de traversées. Les coups de sortie ne sont pas utiles pour ce faire, puisqu'on a vu que les configurations de pile suffisent à simuler les traversées et donc à effectuer la vérification des schémas de récursion. On définit donc les $n$-CPDS, qui n'ont pas de coups de sortie ; l'environnement choisira quelle direction calculer au cours du jeu de parité sur les configurations du $n$-CPDS.

#### 5.2.1. *Les $n$-CPDS et leurs graphes de configuration*

On définit les $n$-CPDS comme suit :

**Définition 30 ($n$-CPDS)** *Un $n$-CPDS est un quadruplet $\mathcal{A} = < \Gamma, Q, \delta, q_0 >$ où $\Gamma$ est l'alphabet de pile (de symbole de fond de pile $\perp$), $Q$ est un ensemble fini d'états, $q_0$ est l'état initial et $\Delta \subset Q \times \Gamma \times Q \times Op_n$ est la relation de transition.*[29]
*On définit les transitions sur les* configurations *$(q, s)$ où $q \in Q$ et $s$ est une $n$-pile ; la configuration initiale est $(q_0, \perp_n)$ où $\perp_n$ est la $n$-pile vide $[\cdots[\perp]\cdots]$. Puisqu'on n'a plus de transitions correspondant à des coups non-internes, il ne reste plus que les transitions $(q, s) \xrightarrow{(q',\theta)} (q', s')$ si et seulement si $s' = \theta(s)$ et $(q, s, q', \theta) \in \Delta$.*

**Définition 31 (Graphe de configurations de $n$-CPDS)** *Etant donné un $n$-CPDS, son* graphe de configurations *est le graphe :*

29. Là encore $\Delta$ ne devra pas mettre ou enlever de symbole $\perp$ sur les piles ; contrairement au cas des $n$-CPDA on autorise ici du non-déterminisme, mais dans la suite il aura seulement lieu au cours du jeu pour le choix des directions par l'environnement, comme auparavant.

*– dont les sommets sont les configurations accessibles,*

*– et dont les arêtes (étiquetées) sont celles induites par la relation de transition* $\xrightarrow{(q',\theta)}$ *sur les configurations accessible.*

**Exemple 14 (Un $2$-CPDS et son graphe de transitions)** *On considère un $2$-CPDS sur l'alphabet de pile $\Sigma = \{\bot, a\}$ et d'états dans $Q = \{0,1,2\}$. Sa relation de transition contient les règles $(0,\bot,1,push_2)$, $(0,a,1,push_2)$, $(1,\bot,2,push_1^{a,2})$, $(1,a,2,push_1^{a,2})$, $(2,a,1,push_2)$, $(2,a,0,collapse)$. Le début de son graphe de configurations est dessiné figure 14.*

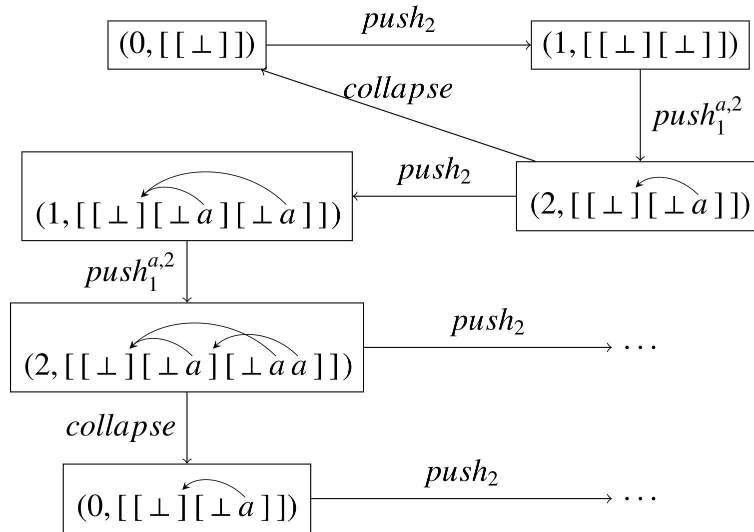



**Figure 4.** *Graphe de transitions d'un $2$-CPDS.*

### 5.2.2. *Jeux de parité pour la vérification sur les graphes de configuration de $n$-CPDS*

On peut définir des jeux de parité sur les graphes de configuration de $n$-CPDS par des techniques similaires à celles qui nous ont permis de passer d'un problème de vérification à un problème d'acceptance d'arbre d'exécution, puis à un jeu d'acceptance à parité. L'ensemble des états est alors partitionné entre Eloïse et Abélard, et induit une partition des configurations ; on étend de même la fonction de priorité des états aux configurations. On obtient une équivalence polynomiale des problèmes suivants :

– Etant donné un jeu de parité sur le graphe de configurations d'un $n$-CPDS, décider si Eloïse a une stratégie gagnante depuis la configuration initiale.

– Etant donné le graphe de configurations d'un $n$-CPDS, décider si une formule de $\mu$-calcul modal $\phi$ est vraie sur la configuration initiale.

– Etant donnés un automate alternant à parité et le graphe de configurations d'un $n$-CPDS, décider si cet automate accepte le déploiement de ce graphe.

On se ramène donc au problème de la résolution de jeux de parité sur des graphes de configuration de $n$-CPDS.

5.2.3. *Esquisse d'une stratégie de résolution sur les jeux de parité de support les graphes de configurations de $n$-CPDS*

Dans (Hague *et al.*, 2008, Section 6), un procédé intéressant de *réduction d'ordre* pour les jeux de parité sur les graphes de configurations de $n$-CPDS est détaillé. A partir d'un jeu de parité sur un $n$-CPDS, on construit un jeu de parité équivalent (sur lequel Eloïse a une stratégie gagnante depuis la configuration initiale si et seulement si elle en a une sur le jeu de parité original) sur un $(n-1)$-CPDS. Une fois encore, nous nous contenterons d'esquisser ici ce procédé dont l'idée générale fut introduite dans (Serre, 2004). Avant de procéder à cette réduction d'ordre, un résultat technique est établi puis utilisé : étant donné un jeu de parité $\mathcal{G}$ sur un graphe de configuration de $n$-CPDS, on peut à l'aide d'une extension de l'alphabet de pile suivie d'une extension cohérente de la relation de transition ajouter aux symboles de pile originaux des informations, de sorte que toute configuration $(q, s)$ obtenue à partir d'une autre par effondrement stockera dans le symbole au sommet de sa pile la priorité minimale rencontrée entre la dernière configuration $(q, s)$ rencontrée et celle qui vient d'être obtenue par effondrement. On dira que ce nouveau jeu de parité $\widehat{\mathcal{G}}$ est *partiellement amnésique* : lorsqu'il s'effondre, il n'oublie pas l'essentiel.
On peut maintenant procéder à la réduction d'ordre à partir de $\widehat{\mathcal{G}}$. Définissons tout d'abord la *hauteur de pile* d'une $n$-pile : il s'agit du nombre de $(n-1)$-piles qui la composent. Dans le jeu $\mathcal{G}'$ que l'on va décrire, on ne pourra jouer que sur une $(n-1)$-pile. Remarquons que, dans la $n$-pile sur laquelle on joue dans $\widehat{\mathcal{G}}$, il y a à tout stade d'une partie parmi les $(n-1)$-piles qui la composent des $(n-1)$-piles auxquelles on n'accèdera plus (on les dira *inertes*), et d'autres auxquelles on accèdera encore au cours de la partie (on les dira *actives*) : il suffit donc de garder la $(n-1)$-pile active la plus profondément enfouie dans la pile de $\widehat{\mathcal{G}}$ comme support de $\mathcal{G}'$ ; on la notera $s_a$. En effet, les piles inertes peuvent être oubliées ; quant aux piles situées au-dessus de $s_a$, on va voir que deux situations se présentent. Etant donnée une partie $\Lambda = v_0 \cdot v_1 \cdots$ sur $\widehat{\mathcal{G}}$, on en extrait un ensemble de positions $v_{i_0}, v_{i_1}, \ldots$ qui sont telles que :

– Ces positions ne violent pas l'ordre des coups dans $\Lambda$ : si $l \leq k$, $v_{i_l}$ apparaît avant $v_{i_k}$ dans $\Lambda$.

– Pour tout $j$, tous les coups apparaissant entre $v_{i_j}$ et $v_{i_{j+1}}$ dans $\Lambda$ sont des configurations de hauteur de pile strictement supérieure à celles de $v_{i_j}$ et de $v_{i_{j+1}}$.

– Pour tout $j$, la hauteur de pile de $v_{i_{j+1}}$ n'est strictement supérieure à celle de $v_{i_j}$ qu'à condition qu'il n'y ait pas dans $\Lambda$ de coup apparaissant après $v_{i_{j+1}}$ et étant de hauteur de pile strictement inférieure.

et on introduit deux termes supplémentaires :

– les *bosses* sont les couples $(v_{i_j}, v_{i_{j+1}})$ dont les deux composantes sont de même hauteur de pile,

– et les *marches* sont les couples $(v_{i_j}, v_{i_{j+1}})$ dont la seconde composante est de hauteur de pile supérieure.

Dans $\mathcal{G}'$, l'idée est de simuler ce qui se passe entre $v_{i_j}$ son successeur $v_{i_{j+1}}$ dans le jeu original sur $\widehat{\mathcal{G}}$. On ne peut en effet maintenir que $s_a$, qui ne peut donc être altérée au cours de la simulation d'une bosse, et le sera lors du franchissement d'une marche ($s_a$ devenant alors inerte, on la remplace par la $(n-1)$-pile active immédiatement au-dessus d'elle, qui jouera dorénavant son rôle). On distinguera donc dans $\mathcal{G}'$ deux situations, correspondant à la simulation en un seul coup d'une suite finie de coups de $\Lambda$ : lorsqu'on a simulé $\Lambda$ jusqu'au coup $v_{i_j}$, on arrive soit :

– sur une bosse $(v_{i_j}, v_{i_{j+1}})$ triviale : $v_{i_j}$ et $v_{i_{j+1}}$ sont consécutifs dans $\Lambda$, il suffit de jouer comme sur $\widehat{\mathcal{G}}$ puisque les opérations ne nécessitent que $s_a$.

– sur une bosse $(v_{i_j}, v_{i_{j+1}})$ non-triviale : Eloïse doit simuler la sous-partie de $\Lambda$ comprise entre $v_{i_j}$ et $v_{i_{j+1}}$. Elle le fait, et retourne en la configuration correspondant à $v_{i_{j+1}}$ sur $\mathcal{G}'$ en apportant également l'information sur la priorité minimale rencontrée au cours de la simulation[30], et l'état de retour ainsi que la pile $s_a$ obtenue à ce stade (ceux-ci devant par construction correspondre à la configuration portée par le coup $v_{i_{j+1}}$). La priorité du coup $v_{i_{j+1}}$ de $\mathcal{G}'$ sera le minimum de celle calculée par Eloïse au cours de la simulation de la bosse et de celle de l'état correspondant à la configuration $v_{i_{j+1}}$. C'est ensuite au tour d'Abélard de jouer : il peut mettre en doute la simulation d'Eloïse, auquel cas celle-ci est effectivement menée (ce qui altère donc définitivement la pile $s_a$) ; Eloïse perd si sa simulation de la bosse était erronée, et Abélard perd s'il a mis en doute à tort la qualité de celle-ci. Sinon, Abélard laisse à Eloïse le soin de simuler le passage du coup $v_{i_{j+1}}$ au coup $v_{i_{j+2}}$.

– sur une marche $(v_{i_j}, v_{i_{j+1}})$ : par construction, $v_{i_j}$ et $v_{i_{j+1}}$ sont deux coups successifs de $\Lambda$, et le second correspond à une configuration de hauteur de pile valant une unité de plus que celle correspondant au premier. Eloïse n'a rien à simuler puisque les deux coups sont successifs, on se contente donc de remplacer $s_a$ par la nouvelle pile active la plus enfouie dans la $n$-pile de la configuration correspondant à $v_{i_{j+1}}$ dans $\widehat{\mathcal{G}}$.

$\mathcal{G}'$ simule bien $\widehat{\mathcal{G}}$ : pour qu'Eloïse ait une stratégie gagnante à partir de la configuration initiale du jeu, elle ne doit en particulier pas effectuer de mauvaise simulation des bosses non-triviales. De plus, les conditions de parité sont équivalentes : sur les simulations de marches et de bosses triviales, $\mathcal{G}'$ et $\widehat{\mathcal{G}}$ ne diffèrent pas et mènent donc à l'apparition des mêmes priorités ; sur les marches non-triviales, la finitude de celles-ci et la propagation de la priorité minimale rencontrée au cours de leur simulation (permise par l'amnésie partielle) assure cette équivalence.

30. C'est ici qu'intervient l'amnésie partielle : en cas d'effondrement, Eloïse peut continuer à propager l'information sur la priorité minimale rencontrée au cours de la simulation de la bosse.

5.2.4. *Application à la complexité*

Tout d'abord, la résolution des jeux de parité sur les graphes de configuration de 1-CPDS est EXPTIME-complète : en effet, ce cas correspond à celui des jeux de parité sur les automates à pile usuels, puisqu'alors l'opération d'effondrement se réduit à l'opération de dépilement *pop* usuelle, et ce cas a été traité dans (Walukiewicz, 2001). Il s'avère de plus que le jeu d'ordre $n-1$ obtenu par réduction d'ordre à partir d'un jeu de parité sur un graphe de configuration de $n$-CPDS donne un jeu exponentiellement plus gros (voir (Hague *et al.*, 2008, Section 6)). Ceci fournit donc une nouvelle preuve du théorème 5. Cette approche fournit également un autre moyen de vérifier les schémas de récursion d'ordre $n$.

REMARQUE. — Arbres générés par des schémas de récursion et théories MSO
MSO et le $\mu$-calcul modal ayant la même expressivité sur les arbres (Janin *et al.*, 1996), il suit du théorème 5 que les arbres générés par les schémas de récursion d'ordre supérieur ont des théories MSO décidables.

## Conclusion et développements ultérieurs

Nous avons présenté dans cette étude les bases de la sémantique des jeux et une application de celle-ci à la révélation de la sémantique de dispositifs de modélisation de programmes sans contrôle ni effets de bord, les schémas de récursion d'ordre supérieur, et ce en vue de leur vérification. L'utilisation du $\mu$-calcul modal, logique à la fois adaptée à la structure arborescente des productions des schémas de récursion et permettant la traduction aisée des problèmes de vérification en jeux de parité, nous a amenés de la sémantique des jeux aux jeux de vérification. Il est à noter qu'avant l'utilisation de la sémantique des jeux dans (Ong, 2006), la question de la complexité de la vérification des schémas de récursion d'ordre supérieur n'avait pas été résolue. Ce résultat ouvre de fait la voie à une interaction intéressante entre deux domaines qui n'ont que peu eu l'occasion à ce jour de se cotôyer.
L'approche suivie dans cette étude et dans les articles sur lesquels elle s'appuie est essentiellement théorique, et n'est que peu effective. Une preuve ultérieure du résultat de complexité se trouve dans (Kobayashi *et al.*, 2009), et utilise plutôt un système de types intersection dont l'effet est de simuler l'exécution d'un automate alternant de parité correspondant à une formule de $\mu$-calcul modal sur l'arbre produit par un schéma de récursion, sans recours à la sémantique des jeux. L'implémentation de cette approche est efficace, voir par exemple (Kobayashi, 2011, Page 85).

## 6. Bibliographie

Abramsky S., Malacaria P., Jagadeesan R., « Full Abstraction for PCF », *in* M. Hagiya, J. C. Mitchell (eds), *TACS*, vol. 789 of *Lecture Notes in Computer Science*, Springer, p. 1-15, 1994.

Abramsky S., McCusker G., « Game Semantics », *in* H. Schwichtenberg, U. Berger (eds), *Computational Logic : Proceedings of the 1997 Marktoberdorf Summer School*, Lecture Notes in Computer Science, Springer-Verlag, p. 1-56, 1999.

Blass A., « A Game Semantics for Linear Logic », *Ann. Pure Appl. Logic*, vol. 56, n° 1-3, p. 183-220, 1992.

Blum W., Ong C.-H. L., « A Concrete Presentation of Game Semantics », *GALOP*, p. 139, 2008.

Blum W., Ong C.-H. L., « Local computation of $\beta$-reduction », , Annals of Pure and Applied Logic (en préparation) - `http://william.famille-blum.org/research/APAL-localbeta.pdf`, 2011.

Bradfield J., Stirling C., « Modal logics and mu-calculi : an introduction », *in* A. P. J. Bergstra, S. Smolka (eds), *Handbook of Process Algebra*, Elsevier, p. 293-330, 2001.

Damm W., « The IO- and OI-Hierarchies », *Theor. Comput. Sci.*, vol. 20, p. 95-207, 1982.

Emerson E. A., Jutla C. S., « Tree Automata, Mu-Calculus and Determinacy », *FOCS*, IEEE, p. 368-377, 1991.

Ghica D. R., Dimovski A., Lazic R., « Abstraction-refinement for game-based model checking », *in* D. R. Ghica, G. McCusker (eds), *GALOP*, p. 139, 2005.

Girard J.-Y., « Linear Logic », *Theor. Comput. Sci.*, vol. 50, p. 1-102, 1987.

Goubault-Larrecq J., « Cours de $\lambda$-calcul », , `http://www.lsv.ens-cachan.fr/~goubault/Lambda/loginfoindex.html`, 2011.

Greenland W., Game Semantics for Region Analysis, PhD thesis, University of Oxford, 2004.

Grellois C., « Modal $\mu$-calculus model-checking games over APT and $n$-CPDS as a consequence of game semantics of higher-order recursion schemes (version avec annexe) », , `http://student.grellois.fr/hors.pdf`, 2010.

Hague M., Murawski A. S., Ong C.-H. L., Serre O., « Collapsible Pushdown Automata and Recursion Schemes », *LICS*, IEEE Computer Society, p. 452-461, 2008.

Harmer R., McCusker G., « A Fully Abstract Game Semantics for Finite Nondeterminism », *LICS*, p. 422-430, 1999.

Hyland J. M. E., Ong C.-H. L., « On Full Abstraction for PCF : I, II, and III », *Inf. Comput.*, vol. 163, n° 2, p. 285-408, 2000.

Igarashi A., Kobayashi N., « Resource usage analysis », *ACM Trans. Program. Lang. Syst.*, vol. 27, n° 2, p. 264-313, 2005.

Janin D., Walukiewicz I., « On the Expressive Completeness of the Propositional mu-Calculus with Respect to Monadic Second Order Logic », *in* U. Montanari, V. Sassone (eds), *CONCUR*, vol. 1119 of *Lecture Notes in Computer Science*, Springer, p. 263-277, 1996.

Joyal A., « Remarques sur la théorie des jeux à deux personnes », *Gazette des Sciences Mathématiques du Québec*, vol. 1, p. 46-52, 1977.

Jurdzinski M., « Small Progress Measures for Solving Parity Games », *in* H. Reichel, S. Tison (eds), *STACS*, vol. 1770 of *Lecture Notes in Computer Science*, Springer, p. 290-301, 2000.

Kobayashi N., « Higher-Order Model Checking : From Theory to Practice (Transparents d'exposé invité à LICS 2011) », , `http://www.kb.ecei.tohoku.ac.jp/~koba/slides/LICS2011.pdf`, 2011.

Kobayashi N., Ong C.-H. L., « A Type System Equivalent to the Modal Mu-Calculus Model Checking of Higher-Order Recursion Schemes », *LICS*, IEEE Computer Society, p. 179-188, 2009.

Kozen D., « Results on the Propositional $\mu$-Calculus », *Theor. Comput. Sci.*, vol. 27, p. 333-354, 1983.

Melliès P.-A., « Asynchronous Games 3 An Innocent Model of Linear Logic », *Electr. Notes Theor. Comput. Sci.*, vol. 122, p. 171-192, 2005.

Ong C.-H. L., « On Model-Checking Trees Generated by Higher-Order Recursion Schemes », *LICS*, IEEE Computer Society, p. 81-90, 2006.

Scott D., Bakker J. D., « A theory of programs », *Unpublished manuscript, IBM, Vienna*, 1969.

Serre O., Contribution à l'étude des jeux sur des graphes de processus à pile, PhD thesis, University of Oxford, 2004.

Walukiewicz I., « Pushdown Processes : Games and Model-checking », *Inf. Comput.*, vol. 164, n° 2, p. 234-263, 2001.

Wilke T., « Alternating tree automata, parity games, and modal mu-calculus », *Bull. Belg. Math. Soc.*, vol. 8, n° 2, p. 359-391, 2002.