

FL-system's Intelligent Cache

Gurvan Le Guernic, Julien Perret

► **To cite this version:**

Gurvan Le Guernic, Julien Perret. FL-system's Intelligent Cache. MajecSTIC 2005: Manifestation des Jeunes Chercheurs francophones dans les domaines des STIC, IRISA – IETR – LTSI, Nov 2005, Rennes, pp.79-88. inria-00000709

HAL Id: inria-00000709

<https://hal.inria.fr/inria-00000709>

Submitted on 15 Nov 2005

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

FL-system's Intelligent Cache

Gurvan Le Guernic, Julien Perret

IRISA, Campus universitaire de Beaulieu,
Avenue du Général Leclerc, 35042 Rennes Cedex - France
{Gurvan.Le_Guernic, Julien.Perret}@irisa.fr

Résumé : Cet article présente une application de techniques issues du génie logiciel à la modélisation d'environnements virtuels. Ces derniers, dès lors qu'ils sont complexes, constituent des masses de données particulièrement volumineuses et, de ce fait, difficiles à manipuler. De plus, les descriptions exhaustives de ces environnements, c'est-à-dire explicites, sont peu évolutives. Pour résoudre ces problèmes, il existe des méthodes basées sur des systèmes de réécriture permettant de décrire de façon générique les objets de l'environnement. Chaque objet est ainsi décrit par un axiome qui est réécrit lors de la génération de l'environnement. Lors de cette phase, il est fréquent de retrouver deux séquences de réécriture aboutissant à un même objet. Il est alors possible d'utiliser un mécanisme tel que le cache afin d'améliorer les performances en profitant des calculs déjà effectués.

Cependant, se contenter de mettre en correspondance les noms et paramètres des axiomes n'est pas suffisant pour établir l'ensemble des réécritures identiques. En effet, les réécritures d'un même axiome avec des paramètres effectifs différents peuvent aboutir au même objet. Le système proposé dans cet article effectue une analyse dynamique des réécritures afin d'améliorer les performances du cache en détectant de tels cas.

La première partie de cet article présente le système de réécriture, la seconde l'analyse utilisée, et la dernière le mécanisme proposé.

Mots-clés : Réalité virtuelle - Systèmes de réécriture - 3D - Cache 3D - Calcul de dépendances - Modélisation et informatique fondamentale

1 INTRODUCTION

La gestion d'environnements virtuels 3D de grande taille comporte des contraintes propres. Elle nécessite le stockage de la description d'un grand nombre d'éléments montrant de fortes similitudes structurelles et apparaissant à l'identique ou presque à plusieurs endroits de la scène (on pourra par exemple penser à une rangée de bancs ou de lampadaires). La description exhaustive d'un environnement urbain dans un langage 3D (e.g. VRML) fige le modèle 3D dans un état unique (ou du moins difficilement maintenable) et nécessite une grande capacité de stockage. Une solution consiste à utiliser le mécanisme de réécriture appelé *FL*-

system [Marvie, 2003, Marvie, 2005]. Ce système, basé sur les *L-systems* [Lindenmayer, 1968, Smith, 1984] et les *grammaires de Chomsky* [Prusinkiewicz, 2001], permet de décrire la structure générale des principaux types d'éléments rencontrés dans l'environnement urbain tels que bâtiments et arbres. La réécriture d'un axiome initial à l'aide de la grammaire associée aux éléments urbains d'un type donné permet la génération automatique d'une entité urbaine de ce type. Cela permet une description compacte et évolutive d'environnements urbains 3D de grande taille. Il est ainsi possible de construire de nombreuses entités différentes d'un même type d'objet en exécutant plusieurs réécritures de l'axiome initial avec des paramètres différents.

L'inconvénient majeur d'une telle méthode est le coût, en temps de calcul, impliqué par la phase de réécriture. Cependant, le surcoût imposé par cette phase peut être réduit. Dans le cas de la génération d'une ville de grande taille, il est probable qu'un même bâtiment¹ apparaisse deux fois en des lieux différents de la scène. Il est encore plus probable d'avoir de multiples occurrences d'une même fenêtre. Il serait donc intéressant de pouvoir récupérer le résultat de réécritures déjà effectuées. Il est bien sûr possible d'ajuster la grammaire pour qu'elle réutilise des éléments 3D. Néanmoins, cette méthode présente l'inconvénient de compliquer la tâche du rédacteur des grammaires et de diminuer la généralité structurelle des règles de réécriture. Une autre solution consiste à utiliser un cache. Chaque fois qu'un axiome à réécrire correspond à une clef du cache, l'objet 3D correspondant à cette clef est directement réutilisé.

Dû à la nature paramétrique des axiomes du *FL-system*, un système simple de reconnaissance des clefs du cache se révèle insuffisant. Par exemple, un axiome générateur de bâtiments simples (bat) peut être défini à l'aide de 3 paramètres : d pour la distance au point d'observation (valant 1 pour un bâtiment très éloigné), n pour le nombre d'étages, et s pour le style du bâtiment. Pour n constant, il est probable que cet axiome génère le même bâtiment quel que soit s lorsque d vaut 1. Ainsi $\text{bat}(1, 8, s_1)$ et $\text{bat}(1, 8, s_2)$ aboutissent au même objet 3D. Seulement, il n'est pas évident pour un cache naïf que $\text{bat}(1, 8, s_1)$ et $\text{bat}(1, 8, s_2)$ doivent être considérés comme équivalents.

¹Nous considérons ici deux bâtiments comme identiques si leurs caractéristiques sont suffisamment proches pour pouvoir utiliser, pour les deux, un même représentant, i.e. un même modèle géométrique.

La méthode que nous proposons utilise des notions et mécanismes similaires à ceux que l'on rencontre dans l'analyse de dépendances entre les entrées et les sorties d'un programme [Pugh, 1989, Abadi, 1996]. Les travaux sur les dépendances sont très nombreux dans le domaine de la sécurité informatique, et de nombreux autres problèmes du génie logiciel ont de forts liens avec l'analyse de dépendance [Abadi, 1999]. Dans notre cas, ce type d'analyse permet par exemple de détecter que lorsque d est égale à 1 le résultat final de la réécriture de l'axiome bat est dépendant uniquement de ses deux premiers paramètres. Ainsi, le système de cache peut détecter automatiquement qu'il doit considérer $\text{bat}(1, 8, s_1)$ et $\text{bat}(1, 8, s_2)$ comme équivalents.

Ce document présente tout d'abord une version simplifiée des *FL-systems*. En section 3, nous introduisons un calcul dynamique des dépendances appliquée aux *FL-systems* permettant d'identifier les paramètres intervenant dans la réécriture d'un axiome. La section suivante présente l'application de ce calcul à la réalisation d'un cache *intelligent* pour la phase de réécriture des *FL-systems*. Enfin, nous concluons en section 5.

2 FL-SYSTEM

Les *FL-systems* permettent de modéliser toutes sortes d'objets géométriques et, de préférence, des objets qui présentent une structure fortement redondante, tels des plantes ou des bâtiments. En effet, basés sur une description grammaticale des objets, les *FL-systems* permettent d'utiliser à bon escient la redondance de l'environnement modélisé. Cette répétition nous permet des descriptions très courtes et génériques de classes d'objets. Une classe d'objets est caractérisée par un ensemble de paramètres. Une instance d'objet est alors obtenue par la définition d'un ensemble de valeurs associées à ces paramètres. Néanmoins, les valeurs données à l'axiome d'un *FL-system* n'ont pas nécessairement d'influence sur toutes les parties de l'objet généré. Il est en effet courant qu'une même partie d'un objet se retrouve à l'identique dans un autre objet de la même classe, malgré des différences importantes dans les valeurs de leurs paramètres (on pourra ici penser aux feuilles de deux arbres de la même espèce). Dans cette section, nous proposons une grammaire et une sémantique pour les règles de réécriture des *FL-systems*.

2.1 Le paradigme d'instanciation d'objets géométriques

En informatique graphique, il est courant de vouloir réutiliser des objets géométriques déjà définis ailleurs, d'une part pour réduire l'espace de stockage, mais aussi le coût d'affichage de ces modèles lors de la navigation. Pour ce faire, Hart [Hart, 1992] propose le paradigme d'*instanciation d'objets géométriques* dans le cadre de la modélisation d'objets fractals. Ainsi, à partir de règles simples, il devient possible de réutiliser le même arbre pour définir une forêt en comportant des centaines, chaque arbre pouvant lui-même utiliser plusieurs fois la

même branche et ainsi de suite. Ce paradigme, aujourd'hui intégré à la plupart des langages de modélisation géométrique (e.g. VRML[Marrin, 1997]), améliore très sensiblement les performances des applications 3D.

Dans le cadre de la modélisation d'environnements naturels, Deussen [Deussen, 1998] propose l'*instanciation approximative* comme une extension du paradigme d'instanciation de Hart. Les objets modélisés procéduralement sont projetés dans l'espace des paramètres de l'axiome et ensuite regroupés par *clusters*. Chaque *cluster* représente alors une instance approximative de tous les objets qui en font partie, réduisant ainsi le nombre d'objets que contient la scène.

Les *FL-systems* permettent l'utilisation de nœuds VRML, et donc l'utilisation du paradigme d'instanciation. Nous proposons ici un mécanisme d'instanciation automatique, implicite, et transparent pour l'utilisateur. Ce mécanisme est basé sur le calcul dynamique des dépendances dans les règles de réécriture (cf. section 3).

2.2 La grammaire utilisée

Comme le montre la figure 1, ces règles sont composées d'une partie gauche (*Axiome*), d'une condition (*Cond*), et d'une partie droite (*Terme*). Nous considérerons ici que la stratégie de réécriture est déterministe et récursive à gauche, contrairement aux *L-systems* dont la réécriture est parallèle. Cette restriction a pour but de rendre déterministe chaque étape de la réécriture d'un axiome. Ceci simplifie la présentation des travaux développés dans cet article, en particulier les théorèmes présentés en section 3. De plus, pour contrôler la profondeur de réécriture, nous utiliserons un paramètre donné à l'axiome. Ainsi, à partir d'un axiome donné, une réécriture engendrera un mot composé de terminaux. Les terminaux, dans les *FL-systems*, sont des fonctions et, plus généralement, des appels à des primitives géométriques quelconques (VRML, OpenGL ou autres). Pour plus de détails sur les stratégies de réécriture pouvant être appliquées sur les *FL-systems*, ainsi que sur la nature des mots générés, voir [Marvie, 2005].

La figure 2 donne un exemple de *FL-system*. L'axiome initialise les paramètres n , δ , l , w et $color$, qui représentent respectivement la profondeur de réécriture, un angle de rotation, la longueur d'un segment de branche, le rayon d'une branche et la couleur des feuilles. Les terminaux `rotateX` et `rotateZ` tournent le repère local autour des axes X et Z respectivement. Le terminal `moveZ` déplace le repère local le long de l'axe Z, tandis que les crochets [et] empilent et dépile le repère local, permettant la création facile de sous-branches. Hormis pour l'axiome **L**, il existe 2 règles de réécriture pour tous les axiomes. La première représente la règle de décomposition de l'axiome à proprement parler, l'autre nous assure que lorsque la profondeur de réécriture est atteinte, l'axiome est remplacé par un terminal.

²La profondeur de réécriture d'un *FL-system* est son âge.

$\omega : true \rightarrow \mathbf{A}(7, 22.5, 10, 1, green)$
 $\mathbf{A}(n, \delta, l, w, color) : n > 0 \rightarrow \mathbf{B}(n-1, \delta, l, w, color) \text{ rotateZ}(\delta \times 5) \mathbf{B}(n-1, \delta, l, w, color) \text{ rotateZ}(\delta \times 5) \mathbf{B}(n-1, \delta, l, w, color)$
 $\mathbf{A}(n, \delta, l, w, color) : n = 0 \rightarrow \epsilon$
 $\mathbf{B}(n, \delta, l, w, color) : n > 0 \rightarrow [\text{rotateX}(\delta) \mathbf{F}(n-1, \delta, l, w, color) \mathbf{L}(n-1, \delta, l, w, color) \mathbf{A}(n-1, \delta, l, w \times 0.707, color)]$
 $\mathbf{B}(n, \delta, l, w, color) : n = 0 \rightarrow \epsilon$
 $\mathbf{F}(n, \delta, l, w, color) : n > 0 \rightarrow \mathbf{F}(n-1, \delta, l, w, color) \mathbf{L}(n-1, \delta, l, w, color) \text{ rotateZ}(\delta \times 5) \mathbf{F}(n-1, \delta, l, w, color)$
 $\mathbf{F}(n, \delta, l, w, color) : n = 0 \rightarrow \text{cylinder}(l, w) \text{ moveZ}(l)$
 $\mathbf{L}(n, \delta, l, w, color) : true \rightarrow [\text{rotateX}(-\delta \times 2) \text{leaf}(l, color)]$

FIG. 2 – *FL-system* pour un buisson modifié à partir d'un *L-system* proposé par Prusinkiewicz[Prusinkiewicz, 1990]

$Val = \mathbb{R}^+$
 $Var ::= [a - z]$
 $N\text{Axiome} ::= [A - Z][a - z]^*$
 $N\text{Terminal} ::= [a - z]^*$
 $V ::= Var \mid Val$
 $Terminal ::= N\text{Terminal} (ExPar^*)$
 $OpCond ::= = \mid >$
 $Cond ::= V OpCond V \mid true$
 $OpPar ::= + \mid - \mid x$
 $ExPar ::= (ExPar OpPar ExPar)$
 $\mid V$
 $Axiome ::= NAxiome (ExPar^*)$
 $Terme ::= Axiome Terme$
 $\mid Terminal Terme \mid \epsilon$
 $RWrule ::= Axiome : Cond \rightarrow Terme$

FIG. 1 – Proposition de grammaire pour les règles de réécriture. *Adésigne une liste d'éléments de type \mathcal{A} .



FIG. 3 – Résultat de la réécriture du *L-system* de la figure 2

L'axiome **A** représente un apex³ à partir duquel sont créées trois branches **B**. Les branches **B** se décomposent en une arête **F**, une feuille **L** et un nouvel apex **A**. Chaque arête **F** est décomposée en une arête, une feuille, et une autre arête, simulant ainsi le phénomène de croissance des branches. Lorsque la profondeur de réécriture est nulle, les arêtes **F** sont remplacées par des cylindres (dont l'axe de rotation est l'axe **Z**) grâce au terminal cylinder. Les feuilles **L** sont, quand à elles, remplacées par des objets géométriques leaf.

La figure 4 montre les différences entre la stratégie de réécriture parallèle des *L-systems* et celle proposée ici pour les *FL-systems*. Néanmoins, en s'assurant que les termes situés en partie droite des règles d'écriture complètes du *FL-system* et du *L-system* équivalent après n d'écritures successives sont identiques. La figure 3 montre une interprétation géométrique de la réécriture de cet exemple.

2.3 La sémantique utilisée

Pour pouvoir utiliser un cache pour les *FL-systems*, nous avons besoin d'une sélection des règles de réécriture déterministe. Nous proposons ici une formalisation de la sémantique utilisée pour la réécriture des *FL-systems*. Tout d'abord, nous définissons \mathcal{R} comme la liste (ordonnée) des règles de réécriture d'un *FL-system* donnée. Soit $\mathbf{A}(v^*)$ un axiome tel que $v^* \in \mathbb{R}^*$. Nous avons alors la sémantique donnée en figure 5.

$$\frac{x \in \text{Terminaux}^* \quad \mathbf{A}(p^*) : c_* \wedge c \rightarrow \mathbf{T} = \text{first}(\mathcal{R}, \mathbf{A}(v^*))}{x \mathbf{A}(v^*) y \rightarrow_{\mathcal{R}} x \text{eval}(\mathbf{T}[v^*/p^*]) y}$$

FIG. 5 – Sémantique des règles de réécriture

Nous définissons $\text{first}(\mathcal{R}, \mathbf{A}(\emptyset))$ la fonction qui retourne la règle de réécriture " $\mathbf{A}(p^*) : c_* \wedge c \rightarrow \mathbf{T}$ " où :

- " $\mathbf{A}(p^*) : c \rightarrow \mathbf{T}$ " = r_i est la première règle de \mathcal{R} s'appliquant (*i.e.* la condition " c " est vraie) à $\mathbf{A}(v^*)$
- c_* est la conjonction des négations des conditions des règles précédant dans \mathcal{R} et s'appliquant à un axiome de nom **A**

$X[a/b]$ retourne le résultat de la substitution dans X de toutes les occurrences de b par a . Ainsi,

³En biologie, apex est le nom donné au sommet d'un organe.

$\mathbf{A}(7, 22.5, 10, 1, \text{green})$	$\mathbf{A}(7, 22.5, 10, 1, \text{green})$
$\mathbf{B}(6, \dots) \dots \mathbf{B}(6, \dots) \dots \mathbf{B}(6, \dots)$	$\mathbf{B}(6, \dots) \dots \mathbf{B}(6, \dots) \dots \mathbf{B}(6, \dots)$
$[\dots \mathbf{F}(5, \dots) \mathbf{L}(5, \dots) \mathbf{A}(5, \dots)] \dots \mathbf{B}(6, \dots) \dots \mathbf{B}(6, \dots)$	$[\dots \mathbf{F}(5, \dots) \mathbf{L}(5, \dots) \mathbf{A}(5, \dots)] \dots [\dots \mathbf{F}(5, \dots) \mathbf{L}(5, \dots) \mathbf{A}(5, \dots)] \dots [\dots \mathbf{F}(5, \dots) \mathbf{L}(5, \dots) \mathbf{A}(5, \dots)]$
$[\dots \mathbf{B}(4, \dots) \dots \mathbf{B}(4, \dots) \dots \mathbf{B}(4, \dots)] \dots \mathbf{B}(6, \dots) \dots \mathbf{B}(6, \dots)$	$[\dots \mathbf{B}(4, \dots) \dots \mathbf{B}(4, \dots) \dots \mathbf{B}(4, \dots)] \dots [\dots \mathbf{B}(4, \dots) \dots \mathbf{B}(4, \dots) \dots \mathbf{B}(4, \dots)] \dots [\dots \mathbf{B}(4, \dots) \dots \mathbf{B}(4, \dots) \dots \mathbf{B}(4, \dots)]$
\dots	\dots

FIG. 4 – Comparaison des stratégies de réécriture entre les *FL-systems* proposées (à gauche) et les *L-systems* (à droite).

$(\mathbf{A}(x+3, y))[(1, 0)/(x, y)]$ rend $\mathbf{A}(1+3, 0)$. La fonction $\text{eval}(T)$ parcourt le terme T et évalue les expressions y apparaissant pour obtenir les paramètres numériques de l'étape de réécriture suivante. Par exemple, $\text{eval}(\mathbf{A}(1+3, 0))$ retourne $\mathbf{A}(4, 0)$.

Exemple 2.1: Un autre exemple de *FL-system*

$$\begin{aligned} \omega : \text{true} &\rightarrow \mathbf{A}(1, 2) \\ \mathbf{A}(m, n) : m = 0 &\rightarrow \epsilon \\ \mathbf{A}(m, n) : n > 0 &\rightarrow \mathbf{A}(m, n-1) \\ \mathbf{A}(m, n) : n = 0 &\rightarrow \epsilon \end{aligned}$$

Étant donné le *FL-system* de l'exemple 2.1, l'axiome $\mathbf{A}(1, 2)$ se réécrit en $\mathbf{A}(1, 1)$ avec $\text{first}(\mathcal{R}, \mathbf{A}(1, 2))$ égal :

$$\mathbf{A}(m, n) : \neg(m = 0) \wedge (n > 0) \rightarrow \mathbf{A}(m, n-1)$$

Dans cet exemple, la condition $m \neq 0 \wedge n > 0$ donne la dépendance vis-à-vis des variables m et n de la règle appliquée. Si la négation de $m = 0$ n'était pas présente, l'importance de la valeur de m ne serait pas explicite. Toujours dans ce même exemple, la fonction eval remplace le paramètre m par sa valeur (1), et évalue l'expression " $n-1$ " à 1.

Le paragraphe précédent aborde la dépendance qui existe entre la valeur des paramètres d'un axiome et le résultat final de sa réécriture. La section suivante propose d'aller plus loin dans l'étude de cette dépendance.

3 CALCUL DYNAMIQUE DES DÉPENDANCES

Le système de cache présenté dans cet article met à profit la propriété de *joignabilité* de certains termes dans un système de réécriture [Dershowitz, 1990]. Si deux termes se réécrivent en un même terme sous forme normale (*i.e.* composé uniquement de terminaux) alors ces deux termes sont dit *joignables*. Dans cette section, nous exposons la méthode employée afin de calculer dynamiquement un ensemble, aussi grand que possible, de termes joignables avec le terme en cours de réécriture. La méthode employée s'inspire de travaux sur la confidentialité de programmes séquentiels [Le Guernic, 2005]. Elle se base sur la modification de la sémantique de réécriture afin de manipuler des *valeurs étiquetées*. La réécriture d'un axiome, paramétré par des *valeurétiquetées*, retourne alors un terme sous forme normale, lui-même marqué par une étiquette. Cette dernière décode des étiquettes des paramètres de l'axiome qui ont influencé cette réécriture.

3.1 Adaptation de la grammaire

La simple modification, donnée en figure 6, de notre grammaire initiale (figure 1) permet d'intégrer les valeurs étiquetées. $\mathcal{P}(\mathbb{L}\text{Label})$ est l'ensemble des ensembles de $\mathbb{L}\text{Label}$. Les valeurs (Val) sont maintenant des paires notées " $e : n$ ". Le deuxième élément de cette paire (n), aussi appelé valeur numérique, appartient à \mathbb{R} (" n " est la valeur utilisée dans la section 2). Le premier élément (e) est un ensemble de *labels* formant une *étiquette*. Chaque étiquette de l'axiome initial est composée d'un unique label. Chacun de ces labels n'est utilisé qu'une seule fois (*cf.* exemple 3.1).

Exemple 3.1: Exemple d'étiquetage initial

$$\mathbf{A}(\{l1\} : 7, \{l2\} : 22.5, \{l3\} : 10, \{l4\} : 1, \{l5\} : \text{green})$$

$$\begin{aligned} \text{Étiquette} &= \mathcal{P}(\mathbb{L}\text{Label}) \\ Val &= (\text{Étiquette} \times \mathbb{R}^+) \end{aligned}$$

FIG. 6 – Grammaire des valeurs étiquetées

3.2 Une nouvelle sémantique

La figure 7 présente la nouvelle sémantique utilisée pour la réécriture de termes dont les valeurs sont étiquetées. Cette sémantique présente de grandes similitudes avec celle présentée dans la section 2. Il est facile de se convaincre que, mis à part les étiquettes, ces deux sémantiques réécrivent un axiome de la même façon.

$$\begin{array}{c} x \in \text{Terminaux}^* \\ \mathbf{A}(p^*) : c_* \wedge c \rightarrow \mathbf{T} = \text{first}(\mathcal{R}, \mathbf{A}(v^*)) \\ e_c = (\text{var}(c_* \wedge c))[\text{lab}(v^*)/p^*] \\ e_{op} = (\text{var}(\text{op}(\mathbf{T})))[\text{lab}(v^*)/p^*] \\ e' = e \cup e_c \cup e_{op} \\ \hline e : x \mathbf{A}(v^*) y \rightarrow_{\mathcal{R}} e' : x \text{eval}(\mathbf{T}[v^*/p^*]) y \end{array}$$

FIG. 7 – Nouvelle sémantique des règles de réécriture

Dans cette nouvelle sémantique, chaque terme est également étiqueté. Cette étiquette identifie l'ensemble des paramètres qui ont influencé la réécriture. Afin de calculer la nouvelle étiquette d'un terme lors de sa réécriture, nous introduisons trois nouvelles fonctions :

op retourne la liste des opérations contenues dans son argument. Par exemple, " $\text{op}(\mathbf{A}(x, x+y, x+1))$ " rend la liste $[x+y, x+1]$.

⁴On appelle axiome initial, l'axiome servant de point de départ à la réécriture.

`var` retourne l'ensemble des variables de son argument. Ainsi "`var(x > y)`" rend en résultat l'ensemble $\{x, y\}$.

`lab` extrait l'étiquette d'une valeur. Par exemple, "`lab(e : 3)`" retourne l'étiquette e .

La fonction `eval` est également modifiée afin de calculer avec des valeurs étiquetées. Lors de l'évaluation d'un paramètre sous forme d'opération, `eval` fait abstraction des étiquettes et effectue l'opération à partir des valeurs numériques. Une nouvelle étiquette vide est attribuée au résultat. Par exemple, `eval(A((e:1) + 3))` retourne $A(\emptyset : 4)$.

Lors de la réécriture d'un terme t dans un terme t' ($e : t \rightarrow_{\mathcal{R}} e' : t'$), la nouvelle étiquette (\hat{e}) est obtenue en faisant l'union des labels de l'ancienne étiquette avec les labels des étiquettes e_i et e_{op} , où :

e_c contient les labels des paramètres influençant la sélection de la règle de réécriture pour cette étape de l'évaluation.

e_{op} contient les labels des paramètres qui sont intervenus dans le calcul d'une opération à cette étape.

3.3 Propriétés de la sémantique

Afin d'énoncer certaines propriétés de la sémantique, une relation de projection entre paramètres est tout d'abord définie. Cette définition s'appuie sur une représentation des paramètres d'un axiome sous la forme d'une liste de valeurs étiquetées. Cette projection s'exprime en fonction d'une étiquette qui restreint les paramètres pris en compte. On note " $P_1 \xrightarrow{e} P_2$ " la projection d'une liste de paramètres P_1 dans une autre liste de paramètres P_2 de façon restreinte à l'étiquette e . Pour toutes listes de paramètres P_1 et P_2 , $P_1 \xrightarrow{e} P_2$ si et seulement si les paramètres, dont l'étiquette dans P_1 est un sous-ensemble de e , ont la même valeur numérique dans P_1 et dans P_2 . Une définition récursive plus formelle de cette notion est donnée dans la définition 3.1.

Définition 3.1 ($P_1 \xrightarrow{e} P_2$)

- $[\] \xrightarrow{e} [\]$ est vrai
- $[e_1 : x \mid RP_1] \xrightarrow{e} [- : y \mid RP_2]$ est vrai ssi :
 1. $x = y$ ou $e_1 \not\subseteq e$
 2. $RP_1 \xrightarrow{e} RP_2$

Trois nouvelles fonctions sont également introduites :

`labellise` prend en argument une liste de valeurs numériques (\vec{n}) et une liste d'étiquettes de même longueur (e^*). Elle retourne une liste de valeurs étiquetées v de même longueur que n^* . v^* est construite de telle sorte que, pour tout i , $v_i = e_i : n_i$. Par exemple, `labellise([e1, e2], [5, 1])` retourne la liste $[e_1 : 5, e_2 : 1]$.

`strip` prend en argument un terme T construit à partir de valeurs étiquetées. Elle retourne un terme T' qui correspond au terme T dont toutes les étiquettes ont été supprimées. Ainsi, `strip(A(e1, e2 : 7))` retourne $A(2, 7)$.

$T[v^*/e^*]^{\mathcal{E}}$ est une fonction de substitution basée sur les étiquettes. Le résultat de la substitution $T[v^*/e^*]^{\mathcal{E}}$ est le terme T dans lequel les valeurs dont l'étiquette apparaît à la position i dans la liste e^* ont été remplacées par la valeur à la position i dans \vec{v} . Par exemple, $A(e_1 : 2, e_2 : 7)[(e_a : 3, e_b : 4)/(e_2, e_3)]^{\mathcal{E}}$ est le terme $A(e_1 : 2, e_a : 3)$.

Théorème 3.1 (Stabilité une étape) ⁵

Pour tout étiquette e_i , axiome A , liste de valeurs numériques n^* et d'étiquettes e_v^* telles que $\text{labellise}(n^*, e_v^*) = v^*$ avec v^* paramètres de A , et ensemble de règles \mathcal{R} , si :

$$e_i : A(v^*) \rightarrow_{\mathcal{R}} e : T$$

alors, pour toute liste de valeurs v_2^* telle que $v^* \xrightarrow{e} v_2^*$:

$$- : A(v_2^*) \rightarrow_{\mathcal{R}} - : T'$$

avec $\text{strip}(T') = \text{strip}(T[v_2^*/e_v^*]^{\mathcal{E}})$.

Le théorème 3.1 stipule que, si un axiome A paramétré par v^* se réécrit en une étape dans le terme T , alors le même axiome, paramétré par une liste de valeurs dans laquelle se projette v^* (restreint par l'étiquette g'en'érée), se réécrit en une étape dans le terme T' . Les étiquettes mises à part, ce terme T' correspond au terme T dans lequel les valeurs étiquetées apparaissant dans v ont été remplacées par les valeurs correspondantes des paramètres utilisés pour la réécriture de T' .

Soit $\rightarrow_{\mathcal{R}}^{\dagger}$ la fermeture transitive de $\rightarrow_{\mathcal{R}}$ aboutissant à un terme sous forme normale. Le corollaire 3.2 est la spécialisation au cas où $e = \emptyset$ de la généralisation du théorème 3.1 à $\rightarrow_{\mathcal{R}}^{\dagger}$.

Corollaire 3.2 (Reconstructibilité) ⁶

Pour tout axiome A , liste de valeurs numériques n^* et d'étiquettes e_v^* telles que $\text{labellise}(n^*, e_v^*) = v^*$, avec v^* paramètres de A , et ensemble de règles \mathcal{R} , si :

$$\emptyset : A(v^*) \rightarrow_{\mathcal{R}}^{\dagger} e : NT$$

alors, pour toute liste de valeurs v_2^* telle que $v^* \xrightarrow{e} v_2^*$:

$$- : A(v_2^*) \rightarrow_{\mathcal{R}}^{\dagger} - : NT'$$

avec $\text{strip}(NT') = \text{strip}(NT[v_2^*/e_v^*]^{\mathcal{E}})$.

3.4 Interprétation des étiquettes résultants de la réécriture d'un axiome

Dans la suite de cet article, afin de simplifier la présentation, les étiquettes attribuées aux paramètres de l'axiome initial⁷ sont indexées sur les entiers. Ainsi, e_1 est l'étiquette du premier paramètre, e_2 celle du deuxième, et ainsi de suite. Chacune de ces étiquettes est

⁵La preuve du théorème 3.1 découle de l'aspect déterministe de la sémantique utilisée. En effet, à chaque étape de réécriture, seul l'axiome le plus à gauche peut être réécrit. De plus, les étiquettes des valeurs intervenant dans l'évaluation de la condition de la règle de réécriture retournée par la fonction `first` font partie de l'étiquette e g'en'érée lors de l'étape de réécriture. Donc toute réécriture en une étape du même axiome avec des paramètres dans lesquels se projettent les paramètres v^* utilise la même règle de réécriture.

⁶La preuve du corollaire 3.2 s'obtient par spécialisation au cas où $e = \emptyset$ du corollaire équivalent où e n'est pas restreint à l'ensemble vide ; ce dernier corollaire étant prouvé par récurrence sur le nombre d'étapes de la réécriture.

⁷On appelle axiome initial, l'axiome servant de point de départ à la réécriture.

composée d'un label unique. Ces labels sont indexés de la même façon que les étiquettes. Pour tout entier strictement positif i , $e_i = \{l_i\}$. L'exemple 3.2 présente l'ensemble \mathcal{R} des règles de réécriture qui sont utilisées dans cette partie.

Exemple 3.2: Ensemble des règles des axiomes **A** et **B**

$$\begin{aligned} \mathbf{A}(c, d, x) &: c = 0 \rightarrow \mathbf{B}(d, x) \\ \mathbf{A}(c, d, x) &: c > 0 \rightarrow v(d+c) \mathbf{A}(c-1, d, x) \\ \mathbf{B}(d, x) &: d = 0 \rightarrow t() \\ \mathbf{B}(d, x) &: d > 0 \rightarrow u(x) \mathbf{B}((d-1), x) \end{aligned}$$

Soit e l'étiquette associée au terme sous forme normale résultant de la réécriture de l'axiome **C** ($\emptyset : \mathbf{C}(v) \rightarrow_{\mathcal{R}}^{\dagger} e : \mathbf{NT}$). e contient les labels de tous les paramètres qui ont influencés la sélection des règles lors de la réécriture de **C**; et donc la structure générée du terme sous forme normale obtenu. Par exemple, l'évaluation de l'axiome $\mathbf{B}(e_1 : 0, e_2 : 3)$ extériorise uniquement l'étiquette de son premier paramètre et rend le terme sous forme normale $t()$. Celui-ci n'ayant pas de terminaux avec paramètre, le corollaire 3.2 implique que tout axiome de la forme $\mathbf{B}(_ : 0, _)$ se réécrit sous forme normale en $t()$. La réécriture de l'axiome $\mathbf{A}(e : 0, e_2 : 0, e_3 : 3)$ fonctionne de façon similaire. Le résultat de sa réécriture est donné dans l'exemple 3.3. L'interprétation du résultat de l'évaluation de l'axiome $\mathbf{A}(e : 1, e_2 : 1, e_3 : 1)$ est plus complexe. Le résultat, donné dans l'exemple 3.3, montre l'extériorisation des étiquettes des deux premiers paramètres uniquement. Il est donc possible d'en déduire que la réécriture sous forme normale de tout axiome de la forme $\mathbf{A}(_ : 1, _ : 1, _)$ rend un terme ayant pour structure générale $v(_ : 2) u(_)$. Plus précisément, le paramètre du terminal v ayant une étiquette vide, le corollaire 3.2 implique que le résultat a pour forme $v(_ : 2) u(_)$. De plus, étant donné que le paramètre de u a pour étiquette celle du troisième paramètre de l'axiome initial, ce même corollaire implique que la réécriture sous forme normale de tout axiome de la forme $\mathbf{A}(_ : 1, _ : 1, _ : X)$ rend un terme ayant pour forme $v(_ : 2) u(_ : X) t()$.

Exemple 3.3: Résultats d'évaluations

Le tableau suivant présente des exemples d'évaluation des axiomes **A** et **B** de l'exemple 3.2.

Axiome	Résultat
$\mathbf{B}(e_1 : 0, e_2 : 3)$	$e_1 : [t()]$
$\mathbf{A}(e_1 : 0, e_2 : 0, e_3 : 3)$	$\{l_1, l_2\} : [t()]$
$\mathbf{A}(e_1 : 1, e_2 : 1, e_3 : 1)$	$\{l_1, l_2\} : [v(\emptyset : 2); u(e_3 : 1); t()]$

4 APPLICATION À UN SYSTÈME DE CACHE

Comme présenté en introduction, le but de ce travail est d'accélérer la réécriture des *FL-systems* présentés en section 2 à l'aide d'un mécanisme de cache appelé *cache*

⁸i.e. abstraction faite des paramètres des terminaux.

⁹On dit qu'une réécriture extériorise une étiquette si cette étiquette se retrouve dans l'étiquette du terme sous forme normale obtenu.

intelligent. Dans un cache standard, la paire (clef,valeur) générée par une réécriture est très simple. La clef correspond à l'axiome initial. Elle inclut le nom de l'axiome et la valeur effective de ses paramètres. La valeur associée à cette clef est le terme normal obtenu lors de la réécriture. Cependant, il arrive qu'un paramètre n'intervienne pas dans la réécriture de l'axiome, soit pour une raison de niveau de détails pris en compte, soit parce que le paramètre intervient uniquement au moment du rendu de l'environnement virtuel. Ainsi, avec les règles proposées dans l'exemple 3.2 et comme indiqué dans la section 3.4, les axiomes $\mathbf{B}(0, 3)$ et $\mathbf{B}(0, 0)$ se réécrivent de la même façon. En utilisant un calcul de dépendance comme celui qui est présenté en section 3, il est possible d'améliorer la couverture de la clef de façon à ce qu'une concordance soit déclenchée dans le cache pour deux axiomes qui ne diffèrent que par la valeur de paramètres n'intervenant pas dans la réécriture de ces axiomes. Cette section présente tout d'abord la méthode de génération des clefs étendues et des valeurs associées. L'utilisation de ces paires (clef,valeur) est ensuite expliquée et prouvée correcte au regard de la sémantique donnée dans la figure 7. Enfin, la question de la gestion des paires (clef,valeur) est abordée.

4.1 Génération des clefs et valeurs

Comme exposé en section 3, à partir de l'évaluation d'un axiome paramétré **A**, il est possible de déterminer automatiquement la forme normale réécrite d'un ensemble \mathcal{A} d'axiomes de même nom (**A**) mais paramétrés par des valeurs différentes. Afin d'utiliser ce résultat dans un système de cache, il reste à définir une méthode pour extraire automatiquement une clef caractérisant l'ensemble \mathcal{A} . Il convient également de déterminer la valeur qui doit être associée à cette clef.

Pour ce faire, deux nouvelles fonctions sont définies :

`gen` prend en argument une liste de valeurs étiquetées v^* et une étiquette e . Elle renvoie une liste de valeurs non-étiquetées. L'élément à la position i est la valeur numérique de v_i si l'étiquette de v_i est un sous-ensemble de e , et $_$ sinon. Par exemple, `gen([\{l_1\}:1, \{l_2\}:1, \{l_3\}:1], \{l_1, l_2\})` renvoie `[1, 1, _]`.

`ffonc` prend en argument un terme sous forme normale **T** dont les terminaux le composant sont paramétrés par des valeurs étiquetées. Elle renvoie un terme sous forme normale **T'** dont les terminaux le composant sont paramétrés par des valeurs non-étiquetées. t_j est le terminal à la position j dans **T**, et t'_j celui à la même position dans **T'**. t'_j est construit à partir de t_j en remplaçant chaque valeur $e : v$ par $\#_i$ si l'étiquette e est composée uniquement du label l_i et v sinon. Par exemple, `ffonc(v(\emptyset:2) u(\{l_3\}:1) t())` renvoie `"v(2) u(\#_3) t()"`.

Lors d'une réécriture sous forme normale d'un axiome **A** paramétré par les valeurs étiquetées v

$$\emptyset : \mathbf{A}(v^*) \rightarrow_{\mathcal{R}}^{\dagger} e : \mathbf{NT}$$

¹⁰ v_i est l'élément à la position i dans v

le mécanisme de cache génère la clef $\mathbf{A}(\text{gen}^*(i; e))$ et y associe la valeur $\text{ffonc}(NT)$. L'exemple 4.1 montre les paires (clef,valeur) générées par les exécutions de l'exemple 3.3.

Exemple 4.1: Mise en cache

En utilisant les évaluations présentées dans l'exemple 3.3, le tableau suivant présente les couples (clef,valeur) mis en cache :

Clef	Valeur
$\mathbf{B}(0, -)$	$\{t()\}$
$\mathbf{A}(0, 0, -)$	$\{t()\}$
$\mathbf{A}(1, 1, -)$	$\{v(2); u(\#_3); t()\}$

4.2 Utilisation du cache

Afin de formaliser l'utilisation du cache, une relation de concrétisation entre listes d'éléments appartenant à $\mathbb{R}^+ \cup \{-\}$ est introduite dans la définition 4.1. Une liste l_c est une concrétisation d'une liste de même longueur l_a , notée $l_c \succ l_a$, si et seulement si tout élément appartenant à \mathbb{R}^+ dans l_a est égal à l'élément à la même position dans l_c .

Définition 4.1 ($l_c \succ l_a$)

- $[] \succ []$ est vrai
- $[x \mid RL_c] \succ [y \mid RL_a]$ est vrai ssi :
 1. $y = x$ ou $y = -$
 2. $RL_c \succ RL_a$

On définit également une fonction notée $T\|\#$. T est un terme sous forme normale et v^* est une liste de valeurs non-étiquetées. Cette fonction substitue dans T les identifiants introduits par la fonction $\text{ffonc}(\#_i)$ par la valeur à la position i dans la liste de valeurs v^* .

Le cache est utilisé de la façon suivante. On considère la réécriture de l'axiome \mathbf{A} paramétré par les valeurs non-étiquetées v_c^* . Si il existe une paire $(\mathbf{A}(v_a^*), T)$ telle que $v_c^* \succ v_a^*$, alors le cache rend directement le terme sous forme normale $T[v_c^*]^\#$. La correction du mécanisme est prouvée par le théorème 4.3 ; lui-même prouvée à l'aide des lemmes 4.1 et 4.2 ainsi que du corollaire 3.2. Le résultat de l'application du mécanisme de cache à la réécriture de l'axiome $\mathbf{A}(1, 1, 5)$, lorsque $\mathbf{A}(1, 1, 3)$ a déjà été réécrit, se trouve dans l'exemple 4.2.

Exemple 4.2: Utilisation du cache

Dans cet exemple, le cache contient les paires (clef, valeur) données dans le tableau de l'exemple 4.1. Entre autre, il contient la paire $(\mathbf{A}(1, 1, -), [v(2); u(\#_3); t()])$. La liste de valeur $(1, 1, 5)$ est une concrétisation de la liste $(1, 1, -)$. Ainsi, pour l'axiome $\mathbf{A}(1, 1, 5)$, le cache rend le terme sous forme normale $[v(2); u(\#_3); t()]$.

Lemme 4.1 (Correction de la clef) Pour toute étiquette e et toutes listes de valeurs étiquetées v_c^* et v_a^* :

$$v_c^* \succ \text{gen}(v_a^*, e) \Rightarrow v_a^* \xrightarrow{e} v_c^*$$

Lemme 4.2 (Correction de la valeur) Pour tout terme sous forme normale NT , liste de valeurs étiquetées v_2^* , et liste d'étiquettes e^* telle que l'étiquette à la position i est composée d'un label unique indicé à la valeur i :

$$\text{strip}(NT[v_2^*/e^*]^\mathcal{E}) = (\text{ffonc}(NT))[v_2^*]^\#$$

Théorème 4.3 (Correction du cache) Pour tout axiome \mathbf{A} , liste de valeurs numériques de longueur k n_1^* et n_2^* , et liste d'étiquettes de longueur k e^* telle que l'étiquette à la position i est composée d'un label unique indicé à la valeur i , si :

1. $\emptyset : \mathbf{A}(\text{labellise}(n_1^*, e^*)) \rightarrow_{\mathcal{R}}^{\dagger} e_f : NT$
2. $\emptyset : \mathbf{A}(\text{labellise}(n_2^*, -)) \rightarrow_{\mathcal{R}}^{\dagger} - : NT'$
3. $n_2^* \succ \text{gen}(\text{labellise}(n_1^*, e^*), e_f)$

alors :

$$\text{strip}(NT') = (\text{ffonc}(NT))[n_2^*]^\#$$

4.3 Gestion du cache

Comme pour tout système de cache, les performances du mécanisme proposé sont fortement dépendantes de la gestion des paires (clef,valeur) par le cache. En effet, si le cache contient trop de paires, le système risque de passer plus de temps à essayer de faire correspondre l'axiome à réécrire avec une clef du cache qu'à réécrire l'axiome en question. La résolution de ce problème s'opère à deux niveaux. Tout d'abord, il convient de limiter le nombre de paires (clef,valeur) candidates à l'insertion dans le cache. Enfin, le remplacement de paires présentes dans le cache par de nouvelles doit s'effectuer intelligemment.

Le mécanisme de génération de clefs en produit énormément à chaque niveau de réécriture d'un axiome, une paire (clef,valeur) est générée. Il convient de limiter cette prolifération en ignorant les paires correspondant à un nombre réduit d'étapes de réécriture. Entre autre, les clefs produites lors d'une réécriture utilisant une règle dont le membre droit est composée exclusivement de terminaux doivent être ignorées. Il est également nécessaire de ne pas prendre en compte les paires produites lors de la réécriture d'axiomes rencontrés exceptionnellement. Une grande part de ce travail peut être effectuée par l'auteur d'un *FL-system* en indiquant les axiomes les plus génériques, et donc intéressant à prendre en compte pour la génération des clefs. Par exemple, un *FL-system* produisant une forêt fera une utilisation intensive de l'axiome générant un arbre. Celui-ci étant, à priori, relativement complexe, il est intéressant de le prendre en compte pour la génération des paires.

Lors de l'insertion d'une paire dans le cache, les méthodes standards utilisées dans le domaine des caches permettent d'améliorer les performances. De plus, le mécanisme introduit dans cet article permet, dans certains cas, d'augmenter les capacités du cache sans augmenter sa taille. En effet, lors de l'entrée dans le cache d'une paire (c, v) , si il existe une paire (c', v') présente dans le cache telle que $c' \succ c$, il est possible de remplacer (c', v') par (c, v) sans perte d'expressivité du cache. En effet, la valeur v' peut être régénérée à partir de v .

$\omega : true \rightarrow \mathbf{I}(30, 6, 22.5, 10, 1, green)$
 $\mathbf{I}(f, n, \delta, l, w, color) : f > 0 \rightarrow \mathbf{A}(n, \delta + rand(-2, 2) * 0.5, l + rand(-10, 10) * 0.1,$
 $w + rand(-2, 2) * 0.5, color + rand(-10, 10) * 0.1) \quad \mathbf{I}(f - 1, n, \delta, l, w, color)$
 $\mathbf{I}(f, n, \delta, l, w, color) : f = 0 \rightarrow \epsilon$

FIG. 8 – *FL-system* pour une forêt de buissons utilisant le *FL-system* de la figure 2 (pour simplifier, nous considérons ici les couleurs comme des réels. *green* est donc une constante réelle.)

4.4 Résultats expérimentaux

Le *FL-system* présentée en figure 8 étend celui de la figure 2. Le paramètre f est ajouté afin de permettre un dimensionnement facile du nombre de buissons engendrés par le *FL-system*. Les paramètres suivants sont les mêmes qu'auparavant, mais ils sont ici utilisés en combinaison avec la fonction $rand(a, b)$ qui délivre un entier compris entre a et b (inclus). Cette dernière introduit ainsi du bruit dans les paramètres des buissons générés. Un prototype utilisant ce *FL-system* a été implémenté en *Prolog*. Il simule la réécriture d'un terme sans cache, avec un cache simple, ou avec le système de cache proposé dans cet article. Chaque clef du cache simple se compose du nom de l'axiome et des valeurs effectives des paramètres de l'axiome ayant servi à la génération de cette clef. La figure 9 donne, en fonction du type de cache utilisé, le temps de réécriture d'un terme correspondant à trente buissons de structure identique mais de formes et de couleurs différentes, *i.e.* des axiomes A ayant des valeurs différentes pour les paramètres δ, l, w et $color$. L'axe des abscisses représente la profondeur de réécriture, c'est-à-dire la valeur du paramètre n donné aux axiomes, ou l'âge des buissons. Le gain en temps de réécriture entre la méthode proposée dans cet article et la méthode sans cache est illustré par la figure 10. La figure 11, quant à elle, montre ce même gain, mais pour des réécritures de profondeur de 10 et un nombre de buissons variant de 10 à 30.

On constate sur les figures 9 et 10 que, plus la profondeur augmente, plus notre système de cache s'avère efficace. En effet, par rapport à la réécriture sans cache, la méthode proposée ici profite alors de l'augmentation du nombre de termes *joignables*¹¹ rencontrés lors de la réécriture d'un buisson, c'est-à-dire de la répétition dans sa structure. La figure 11 permet de constater que la grande généralité des clefs produites permet de généraliser certains buissons directement à partir du cache, c'est-à-dire de tirer partie de l'augmentation de la probabilité de retrouver des objets joignables lorsque la taille de l'environnement augmente. Ceci n'est pas possible avec le cache simple. Par contre, lors d'une réécriture ne rencontrant pas de termes aboutissant à la même liste d'appel de primitives graphiques, notre système de cache est alors équivalent au cache simple, voire légèrement plus lent. De fait, les efforts de généralisation des clefs seraient alors une tâche vaine.

¹¹Deux termes sont dits *joignables* si leur réécriture aboutit au même terme sous forme normale

Enfin, il est à noter que la gestion des mécanismes de base du cache par le prototype est très mauvaise. Ceci est reflété par les mauvais résultats du cache simple. Le prototype de cache intelligent dispose du même encodage des mécanismes de base, ce qui laisse à penser qu'une meilleure implémentation permettrait d'obtenir de bien meilleurs résultats.

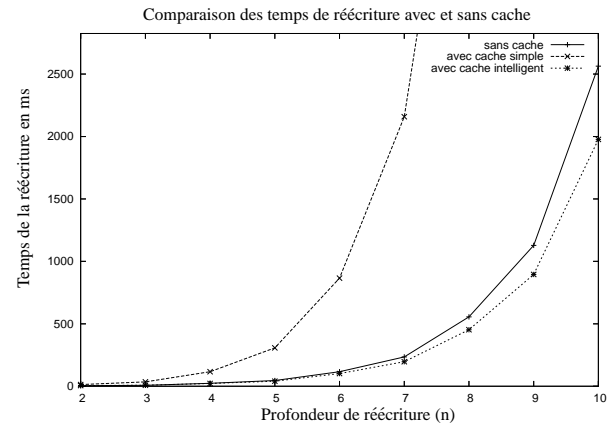


FIG. 9 – Temps de réécriture pour un nombre de buissons $f=30$ et une profondeur de n variant de 2 à 10.

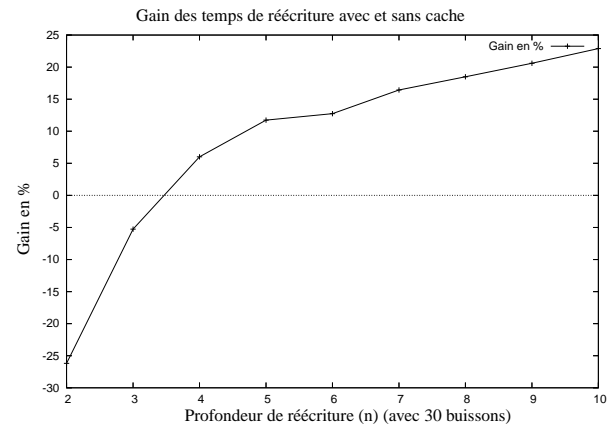


FIG. 10 – Gain en temps de la réécriture avec cache intelligent pour l'expérience de la figure 9

5 CONCLUSION

Dans cet article, nous avons présentée un système de cache *intelligent* basé sur un calcul dynamique des dépendances. Ce nouveau système de cache pour les *FL-systems* permet une abstraction dans le processus de

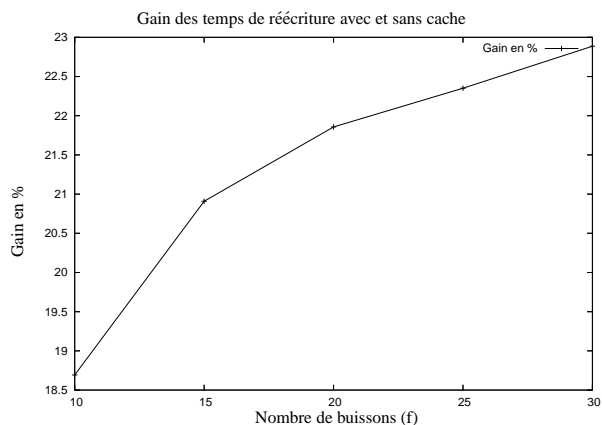


FIG. 11 – Gain de la r  criture pour une profondeur de $n=10$ et un nombre de buissons f variant de 10    30.

mod  lisation g  om  trique : la r  utilisation des objets g  om  triques, jusqu'ici explicite, devient implicite. Cette technique est plus efficace qu'un cache simple gr  ce    la plus grande g  n  ricit   des paires (clef,valeur) produites. Il est ainsi possible de g  n  rer la forme normale d'un axiome qui n'a pas encore   t   rencontr  e par instanciation de la valeur associ  e    une clef *proche* de l'axiome   valuer. N  anmoins, il est important de ne pas calculer des paires (clef,valeur) pour chaque r  gle de r  criture, sous peine de faire d  border le cache. Une solution    ce probl  me consiste    identifier, lors de l'  criture d'un *FL-system* ou automatiquement par analyse, les r  gles les plus utilis  es.

Notre syst  me de cache permet une plus grande souplesse dans la mod  lisation d'environnements tr  s complexes. En effet, consid  rons un *FL-system* repr  sentant un arbre et poss  dant un param  tre *color* stipulant la couleur des feuilles de celui-ci. Une fois cet arbre utilis  e pour mod  liser une for  t, la modification de la valeur du param  tre *color* pourra   tre prise en compte de fa  on automatique par le cache. L'objet repr  sentant une feuille sera alors modifi  e pour prendre en compte le changement de couleur, et le reste de la for  t changera de couleur par la m  me occasion, sans modifier le cache et sans n  cessiter la moindre r  criture.

Enfin, le *FL-system Intelligent Cache* constitue une optimisation compl  mentaire    l'instanciation approximative [Deussen, 1998]. En effet, notre technique permet de d  terminer quels sont les termes dont les r  critures sont identiques, tandis que l'instanciation approximative permet de regrouper des termes similaires gr  ce    leur proximit   dans l'espace des valeurs des param  tres. Notre technique permettrait donc d'am  liorer le *clustering* en identifiant les param  tres qui n'influencent pas la r  criture. En effet, ceci revient    r  duire la dimension de l'espace des param  tres et, de ce fait,    faciliter le *clustering*. Le couplage de ces deux techniques constitue donc une perspective int  ressante pour nos prochains travaux.

BIBLIOGRAPHIE

- [Abadi, 1999] Abadi M., Banerjee A., Heintze N. et Riecke J. G., A core calculus of dependency. *Proceedings of the ACM Symposium Principles of Programming Languages*, pages 147–160.
- [Abadi, 1996] Abadi M., Lampson B. et L'  vy J.-J., Analysis and caching of dependencies. *Proceedings of the 1996 ACM International Conference on Functional Programming*, pages 83–91. ACM Press.
- [Dershowitz, 1990] Dershowitz N. et Jouannaud J.-P., Rewrite systems. In van Leeuwen J., editor, *Handbook of Theoretical Computer Science, Volume B : Formal Models and Semantics (B)*, pages 243–320. North-Holland, Amsterdam.
- [Deussen, 1998] Deussen O., Hanrahan P., Lintermann B., M  ch R., Pharr M. et Prusinkiewicz P., Realistic modeling and rendering of plant ecosystems. *SIGGRAPH '98 : Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 275–286, New York, NY, USA. ACM Press.
- [Hart, 1992] Hart J. C., The object instancing paradigm for linear fractal modeling. *Proceedings of the conference on Graphics interface '92*, pages 224–231. Morgan Kaufmann Publishers Inc.
- [Le Guernic, 2005] Le Guernic G. et Jensen T., Monitoring information flow. In Sabelfeld A., editor, *Proceedings of the 2005 Workshop on Foundations of Computer Security*. DePaul University. LICS'05 Affiliated Workshop.
- [Lindenmayer, 1968] Lindenmayer A., Mathematical models for cellular interactions in development, I & II. *Journal of Theoretic Biology*, 18 :280–315.
- [Marrin, 1997] Marrin C., Carey R. et Bell G., A vrml specification. Rapport technique, VRML consortium. <http://www.vrml.org/Specifications/VRML97>.
- [Marvie, 2003] Marvie J.-E., Perret J. et Bouatouch K., Remote interactive walkthrough of city models. *Proceedings of Pacific Graphics*, volume 2, pages 389–393.
- [Marvie, 2005] Marvie J.-E., Perret J. et Bouatouch K., Fl-system : A functional l-system for procedural geometric modeling. *The Visual Computer*. To appear.
- [Prusinkiewicz, 1990] Prusinkiewicz P., Lindenmayer A., Hanan J. S. et al., *The algorithmic beauty of plants*. Springer-Verlag, New York.
- [Prusinkiewicz, 2001] Prusinkiewicz P., Mundermann L., Karwowski R. et Lane B., The use of positional information in the modeling of plants. *Proceedings of Computer Graphics and Interactive Techniques*, pages 289–300. ACM Press.
- [Pugh, 1989] Pugh W. et Teitelbaum T., Incremental computation via function caching. *Conference Record of the 16th Annual Symposium on POPL*, pages 315–328.
- [Smith, 1984] Smith A. R., Plants, fractals, and formal languages. *Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, pages 1–10. ACM Press.