FLIC: Application to Caching of a Dynamic Dependency Analysis for a 3D Oriented CRS

Gurvan Le Guernic, Julien Perret

FLIC: Application to Caching of a Dynamic Dependency Analysis for a 3D Oriented CRS

Gurvan Le Guernic [1]

KSU, Manhattan, KS 66506, USA
&
IRISA, 35042 Rennes, France

Julien Perret [2]

INRIA Rocquencourt, 78153 Le Chesnay, France
&
City Modeling, virtual research group

Abstract

FL-systems are conditional rewriting systems. They are used for programming (describing) and evaluating (generating) huge 3D virtual environments, such as cities and forests. This paper presents a formal semantics and a dynamic dependency analysis for FL-systems. This analysis allows the characterization of a set of terms which are joinable with the currently rewritten term. Consequently, it is possible to speed up the rewriting steps of the environments generation by using a cache mechanism which is smarter than standard ones. This work can be seen as a dynamic completion of a set of rewriting rules. This completion increases the number of terms which are rewritten in normal form by the application of a single rewriting rule.

Keywords: dependency analysis, conditional rewriting system, FL-system, cache

1 Introduction

Managing huge 3D virtual environments involves dealing with some particular constraints. It requires storing the description of many objects having structural similarities and appearing identically (or nearly so) in different places of the environment. For example, in a virtual city, the geometric objects representing background buildings of the same size and of similar architectural styles are likely to be identical. The exhaustive description of a virtual environment in a 3D modeling language (e.g. VRML [7]) represents a huge amount of data (which can be problematic when this description must be sent to another user, stored, or simply displayed) and implies a difficult management of the environment. With the example of an urban

[1] Email: Gurvan.Le_Guernic@irisa.fr
[2] Email: Julien.Perret@inria.fr

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

environment, changing the style of the street lamps requires going through the entire environment and changing the description of each street lamp. This, obviously, is a lot of work. Fortunately, most 3D modeling languages include the concept of prototype, allowing the user to define parameterized 3D objects. Nevertheless, this technique involves additional work for the user during the modeling process.

Another solution consists in using a conditional rewriting system (CRS) ([4,3]) called *FL-system* [14], short for *Functional Lindenmayer-system*. This system, based on *L-systems* [12] and *Chomsky grammars* [16], enables the description of the general structure of each type of object appearing in the environment. Each object is then generated by rewriting a parameterized module (a level 1 term). The whole environment is described by an axiom (the initial module) which is the starting point of the environment's generation process, and a set of rewriting rules describing the construction of the different types of objects. This method allows for a smaller description of the environment and is easier to manage. The generation of many objects belonging to the same type is achieved by rewriting the same module but with different parameter values.

The main problem of FL-systems, compared to an exhaustive description, is the cost of the rewriting step. However, this cost can be reduced by reusing the result of previous rewriting processes. For example, in an urban environment, it is likely that two buildings appearing in different places will be represented by the same geometric object. It is even more likely that a geometric object representing a window will be reused in a lot of different places. It would then be interesting to be able to directly reuse a geometric object without going through the rewriting step if two modules representing elements of the environment are rewritten to the same geometric object. It is possible to implement this directly in the description of the environment. However this method complicates the work of the author of the FL-system and reduces the generality of the model obtained. Another solution consists in using a cache. Whenever a module has to be rewritten, if a key in the cache maps that module then the geometric object associated to that key is used as the result of the rewriting process.

Due to the parametric nature of the FL-system's modules, a simple syntactic mechanism to verify if a key maps a module is insufficient. For example, a module (of name $\mathbf{B}$) generating a simple building can be defined using three parameters: d for the distance between the building and the observation point (1 being the value for the background), n for the number of floors, and s for the style of the building ($\mathbf{B}(d, n, s)$). For n constant, it is likely that this module would generate the same geometric object whenever d equals 1. Then, $\mathbf{B}(1, 8, s_1)$ and $\mathbf{B}(1, 8, s_2)$ generate the same geometric object. However a simple cache mechanism is not able to detect that both modules must be assumed equivalent.

The method proposed in this paper uses some notions and mechanisms similar to those used for the analysis of dependencies between software inputs and outputs [17,2]. Many computer problems are related to dependency [1]. In our case, a dependency analysis can determine that, whenever d equals 1, the result obtained by rewriting a module $\mathbf{B}(1, \ldots)$ depends only on its two first parameters. This way, the cache mechanism can automatically detect that $\mathbf{B}(1, 8, s_1)$ and $\mathbf{B}(1, 8, s_2)$ must be assumed equivalent. This paper has been inspired by a work of Abadi,

Lampson and Lévy. In [2], they present a dependency analysis for the λ-calculus. This analysis enables them to develop a cache mapping some λ-terms to their value. The work presented here adapts and extends their work to the context of conditional rewriting systems. It can be seen as the dynamic completion of the set of rewriting rules in order to be able to rewrite some particular modules by the application of a single rewriting rule.

This paper starts by presenting a simplified version of FL-systems. In Sect. 3, the dynamic dependency analysis used is formalized. Some properties of this analysis, relevant to the cache system, are stated in the same part. The following section presents the application of the theory to the practical case of the cache mechanism. Finally, related works and conclusion come in Sect. 5.

2 What Are FL-systems?

As stated before, an FL-system is a conditional rewriting system. It is a structural description of the geometric objects appearing in the environment. Those descriptions are shorter than standard descriptions of 3D environments. They are also more generic and allow an easier reuse of previous work. This section proposes a grammar and a semantics for FL-systems.

2.1 A Grammar for FL-systems

A FL-system is a set of rewriting rules. As defined in Fig. 1, a rule ($RWrule$) is composed of a left part which is a module, a condition on the module's variables, and a right part which is a term. The left part is constructed from a module name (its name) and a list of variables or parameters ($\overline{VarName}$). A term is a sequence of modules and terminals. A module is composed of a module name and a list of expressions ($\overline{Exp}$). Constants used in expressions belong to $\mathbb{R}$. In the case of FL-systems, terminals are primitive calls. More precisely, they are calls to geometric primitives belonging to a geometric language or a graphics library (VRML[7,13], OpenGL[18], ...).

$$
\begin{aligned}
Terminal &::= TerminalName\ (\ \overline{Exp}\ ) \quad | \quad Symbols \\
Cond &::= true \quad | \quad Exp = Exp \quad | \quad Exp > Exp \quad | \quad \dots \\
Exp &::= VarName \quad | \quad Constant \\
&\quad | \quad (Exp + Exp) \quad | \quad (Exp \times Exp) \quad | \quad \dots \\
Module &::= ModuleIdentifier\ (\ \overline{Exp}\ ) \\
Term &::= Module\ \ Term \quad | \quad Terminal\ \ Term \quad | \quad \epsilon \\
RWrule &::= ModuleIdentifier\ (\ \overline{VarName}\ ) : Cond \rightarrow Term
\end{aligned}
$$

Fig. 1. Grammar of FL-systems. $\overline{A}$ stands for a list of elements of type A.

Example 2.1 [FL-system of bushes]

$$
\begin{aligned}
\mathbf{A}(n,\delta,l,w,color) \ : n > 0 \rightarrow\ & \mathbf{B}(n-1,\delta,l,w,color) \quad \mathsf{rotateZ}(\delta \times 5) \quad \hookleftarrow \\
& \mathbf{B}(n-1,\delta,l,w,color) \quad \mathsf{rotateZ}(\delta \times 5) \quad \mathbf{B}(n-1,\delta,l,w,color)
\end{aligned}
$$

$$\mathbf{A}(n,\delta,l,w,color) \ : n \leq 0 \rightarrow \epsilon$$

$$\mathbf{B}(n,\delta,l,w,color) \ : n > 0 \rightarrow [\ \mathsf{rotateX}(\delta) \quad \mathbf{F}(n-1,\delta,l,w,color) \quad \mathbf{L}(n-1,\delta,l,w,color) \quad \mathbf{A}(n-1,\delta,l,w\times 0.707,color)]$$

$$\mathbf{B}(n,\delta,l,w,color) \ : n \leq 0 \rightarrow \epsilon$$

$$\mathbf{F}(n,\delta,l,w,color) \ : n > 0 \rightarrow \mathbf{F}(n-1,\delta,l,w,color) \quad \mathbf{L}(n-1,\delta,l,w,color) \quad \mathsf{rotateZ}(\delta\times 5) \quad \mathbf{F}(n-1,\delta,l,w,color)$$

$$\mathbf{F}(n,\delta,l,w,color) \ : n \leq 0 \rightarrow \mathsf{cylinder}(l,w) \quad \mathsf{moveZ}(l)$$

$$\mathbf{L}(n,\delta,l,w,color) \ : true \rightarrow [\ \mathsf{rotateX}(-\delta\times 2) \quad \mathsf{leaf}(l,color)\ ]$$

Example 2.1 is an FL-system which is a modified version of an L-system found in [15]. It is the description of bushes. In this example, the parameters are n, δ, l, w and *color*. n reflects the age of the bush and thus influences its size. δ is a rotation angle which has an effect on the general structure of the bush. l and w give the length and width of each bough. Finally, *color* corresponds to the color of the bush's leaves. Terminal rotateX (respectively rotateZ) rotates the local coordinate system around the x-axis (respectively z-axis). Terminal moveZ moves the local coordinate system along the z-axis; while terminal [(respectively]) pushes (respectively pops) information about the local coordinate system in (respectively from) a stack.

A module $\mathbf{A}(\ldots)$ is an apex from which three boughs are created. A bough $\mathbf{B}(\ldots)$ is composed of a flange $\mathbf{F}(\ldots)$, a leaf $\mathbf{L}(\ldots)$, and a new apex $\mathbf{A}(\ldots)$. Each flange is built from a smaller flange, a leaf, and an other flange. When the recursion stops ($n = 0$), the remaining flanges are replaced by a cylinder of size l and w using the terminal $\mathsf{cylinder}(l,w)$. By increasing the recursion in the generation of flange (which is controlled by the parameter n) and the size of the smaller flanges (l and w), it is possible to simulate the bush's growth phenomenon. Leaves are replaced by geometric objects generated by the terminal $\mathsf{leaf}(l,color)$. Figure 2 shows the graphical result for the module $\mathbf{A}(7,22.5,10,1,green)$.

Fig. 2. Bush resulting from the FL-system of Example 2.1

2.2 A Semantics for FL-systems

In this paper, we assume that an FL-system is *valid* only if any module, whatever the value of the parameters, can be and is rewritten to a unique normal form. A term in normal form is a term composed only of terminals. We also assume that

there is no overloading of the module names. This means that a module name is always followed by the same number of parameters.

The evaluation of an FL-system proceeds in a left to right fashion and enforces a deterministic selection of the rewriting rule applied. This is different from the strategy used for L-systems [12] which is parallel. This restriction is applied in order to make each rewriting step deterministic. The construction of a cache system for the rewriting process only requires that the normal form obtained by rewriting a given module is unique. However, a stronger restriction has been chosen in order to simplify the proofs of the theorems in Sect. 3 as well as their presentation. In the general case, different rewriting strategies can be used for FL-systems. For more details, see [14].

Let $\mathcal{S}em_0$ be the default semantics for FL-systems and $\mathcal{R}$ be the ordered list of the rewriting rules of the FL-system used. $\mathbf{NT}$ is a term in normal form. $\mathbf{A}(\overline{v})$ is a module of identifier $\mathbf{A}$ and whose parameter $\overline{v}$ is a list of values (belonging to $\mathbb{R}$). $\boldsymbol{S}$ is a term. In order to rewrite the term "$\mathbf{NT}\ \mathbf{A}(\overline{v})\ \boldsymbol{S}$", the semantics $\mathcal{S}em_0$ applies the following rule:

$$\frac{\texttt{first}(\mathcal{R}, \mathbf{A}(\overline{v})) \ = \ \mathbf{A}(\overline{p}) : c_* \wedge c \rightarrow \boldsymbol{T}}{\mathbf{NT}\ \mathbf{A}(\overline{v})\ \boldsymbol{S} \quad \rightarrow_{\mathcal{R}} \quad \mathbf{NT}\ \texttt{eval}(\boldsymbol{T}[\overline{v}/\overline{p}])\ \boldsymbol{S}}$$

$\texttt{first}(\mathcal{R}, \mathbf{A}(\overline{v}))$ returns the rewriting rule "$\mathbf{A}(\overline{p}) : c_* \wedge c \rightarrow \boldsymbol{T}$" where: "$\mathbf{A}(\overline{p}) : c \rightarrow \boldsymbol{T}$" is the first rule ($r_i$) in the list $\mathcal{R}$ applying to $\mathbf{A}(\overline{v})$ (i.e. the first rule in $\mathcal{R}$ whose condition c is true with the values $\overline{v}$); and c_* is the conjunction of the negation of the condition of all the rules applying to a module of identifier $\mathbf{A}$ and appearing before r_i in the list $\mathcal{R}$. $\boldsymbol{X}[\overline{v}/\overline{p}]$ returns the term $\boldsymbol{X}$ where all occurrences of a parameter p appearing in $\overline{p}$ are replaced by the value corresponding to p in $\overline{v}$. For example, $(\mathbf{A}(x+3, y))[(1,0)/(x,y)]$ returns $\mathbf{A}(1+3, 0)$. The function $\texttt{eval}(\boldsymbol{T})$ goes through the term $\boldsymbol{T}$ to replace every expression by its value. $\texttt{eval}(\mathbf{A}(1+3,0))$ returns $\mathbf{A}(4,0)$.

2.3 Example of a Rewriting Step

Example 2.2 (A simple FL-system)

$$\mathbf{A}(m,n) : m \leq 0 \rightarrow \mathtt{t}(n)$$

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

$$\mathbf{A}(m,n) : n \leq 0 \rightarrow \mathtt{u}(m)$$

Using the set of rules (FL-system) given in Example 2.2, when rewriting $\mathbf{A}(1,2)$ the function call "$\texttt{first}(\mathcal{R}, \mathbf{A}(1,2))$" returns:

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

The term on the right side of the rule ($\mathbf{A}(m, n-1)$) is then processed. The function $\texttt{eval}$ replaces m by its value (1) and evaluates the expression "$n-1$" to 1. So, one rewriting step applied to $\mathbf{A}(1,2)$ returns $\mathbf{A}(1,1)$. In this example, the condition $\neg(m \leq 0) \wedge n > 0$ emphasizes the dependency between the rule applied and the value of the parameters m and n. If the negation of the condition $m \leq 0$ was not

included in the result of the call to `first`, then the importance of the value of the parameter m would not be explicit.

The preceding paragraph introduces some notions about the dependencies between the result of a rewriting step and the values of the parameters of the module rewritten. The next section formalizes the analysis used to extract those dependencies.

3 A Dynamic Dependency Analysis for CRS

The cache system presented in this paper is based on the notion of *joinability* of terms in rewriting systems [4]. Two terms are said to be joinable if the terms in normal form obtained after rewriting them are the same. For the work presented in this paper, this notion is extended to a notion of *structural joinability*. Two terms are said to be *structurally joinable* if the terms in normal form obtained after rewriting them have the "same structure". Two terms are said to have the "same structure" if they are constituted of the same sequence of modules or terminals, but with parameters which may have different values. For example, $\mathtt{u}(1)\ \mathtt{t}(2,3)$ and $\mathtt{u}(3)\ \mathtt{t}(0,0)$ have the "same structure". Based on this notion, an equivalence relation on terms, denoted $\equiv_S$, is defined.

Definition 3.1 [Same Structure Equivalence Relation] $\boldsymbol{T}$ and $\boldsymbol{S}$ have the same structure, written $\boldsymbol{T} \equiv_S \boldsymbol{S}$, if and only if $\boldsymbol{T}$ and $\boldsymbol{S}$ are empty terms; or the first element of $\boldsymbol{T}$ and the first element of $\boldsymbol{S}$ have the same identifier and the rest of $\boldsymbol{T}$ and $\boldsymbol{S}$ have the same structure.

This section formalizes the method used in order to compute dynamically a set, as big as possible, of terms *structurally joinable* with the term currently rewritten. The method used is inspired by some work on confidentiality [10]. It is based on the modification of the semantics in order for it to manipulate labeled values. Rewriting a module parameterized with labeled values will then return a term in normal form and a label. This label depends on the labels of the parameters which have influenced the structure of the resulting term.

3.1 Computing the Resulting Label

The grammar of FL-systems is slightly modified in order to deal with labeled values. A label is a set of identifiers. Values are now pairs written "$l : n$" and belonging to $\mathbb{L}abel \times \mathbb{R}$. The second element of the pair (n), also called numerical value, corresponds to the value used in Sect. 2. The first element of the pair (l) is a label. At the beginning of a rewriting process, a new label composed of a unique identifier is associated to the value of each parameter of the axiom (the initial module serving as starting point for the rewriting process).

Updated Semantics.

$\mathcal{S}em_0$, presented in Sect. 2.2, is updated in order to deal with labeled values and labeled terms. The identifiers contained in a term's label refer to the parameters which have influenced the rewriting process so far. In order to compute this label,

two new functions are introduced: `var` returns the set of all the variables contained in its argument; and `ident` returns the label (which is a set of identifiers) of its argument. For example, "$\texttt{var}(x > y)$" returns the set $\{x, y\}$ and "$\texttt{ident}(l : 3)$" returns l. When rewriting a term $\boldsymbol{T}$ into the term $\boldsymbol{S}$ ($l : T \rightarrow_{\mathcal{R}} l' : S$) a new label l' is generated. This new label is obtained by union of the previous label l and of a new label l_{c}. l_{c} is the union of the labels of all the parameters which have influenced the selection of the rewriting rule applied. Let us call $\mathcal{S}em_1$ this new semantics. When rewriting the term "$\mathbf{NT}\ \mathbf{A}(\overline{v})\ \boldsymbol{S}$", $\mathcal{S}em_1$ applies the following rule:

$$\frac{\texttt{first}(\mathcal{R}, \mathbf{A}(\overline{v})) \ = \ \mathbf{A}(\overline{p}) : c_* \wedge c \rightarrow \boldsymbol{T} \qquad l_{\mathrm{c}} = \bigcup_{x \in \texttt{var}(c_* \wedge c)} x[\texttt{ident}(\overline{v})/\overline{p}]}{l :\ \mathbf{NT}\ \mathbf{A}(\overline{v})\ \boldsymbol{S} \quad \rightarrow_{\mathcal{R}} \quad l \cup l_{\mathrm{c}} :\ \mathbf{NT}\ \texttt{eval}(\boldsymbol{T}[\overline{v}/\overline{p}])\ \boldsymbol{S}}$$

This new semantics has lots of similarities with $\mathcal{S}em_0$. It can easily be proved that, except for the new labels, both semantics rewrite a module to the same normal form. The proof relies on the fact that derivations in $\mathcal{S}em_0$ and $\mathcal{S}em_1$ are in 1-1 correspondence. With x a member of $\overline{p}$, thus a parameter of the module whose identifier is $\mathbf{A}$, and $\overline{v}$ the list of values applied to the list of parameters $\overline{p}$ of the module whose identifier is $\mathbf{A}$, $x[\texttt{ident}(\overline{v})/\overline{p}]$ is the label of the value applied to the parameter of name x in the call $\mathbf{A}(\overline{v})$. Therefore l_{c} is the union of the labels of the values which are used in the evaluation of $c_* \wedge c$.

As in Sect. 2, the function `eval` evaluates all the expressions contained in its parameter. Those expressions are now composed of labeled values. The labeled value of a constant c is "$\emptyset : c$". For the evaluation of expressions, the function `eval` uses the following rule:

$$\frac{x \ \rightarrow_{\mathrm{e}}\ l_x : n_x \qquad y \ \rightarrow_{\mathrm{e}}\ l_y : n_y}{(x\ op\ y) \quad \rightarrow_{\mathrm{e}} \quad l_x \cup l_y :\ f_{op}(n_x, n_y)}$$

f_{op} is the function corresponding to the operator op. For example, if op is $+$ then f_{op} is the addition and $f_{op}(1, 2)$ is 3.

Properties of the Semantics.

The projection of a list of values $\overline{v}$ into another list of values $\overline{w}$ restrained by the label l is denoted "$\overline{v} \overset{l}{\rightharpoonup} \overline{w}$". For all label l and lists of values $\overline{v}$ and $\overline{w}$, $\overline{v} \overset{l}{\rightharpoonup} \overline{w}$ if and only if any value whose label in $\overline{v}$ is a subset of l has the same numerical value in $\overline{v}$ and $\overline{w}$. A formal definition of this relation follows. In this definition, $[a \mid \overline{b}]$ is the list resulting from the concatenation of a at the beginning of the list $\overline{b}$. Additionally, in this definition, and the remaining of the paper, $\bot$ is used in place of elements which are of no use for stating the definition. For example, as every value has a label, in "$\bot : y$" $\bot$ stands for the label of y; which however is not used in the definition. This projection will be used in the cache mechanism to determine if a result previously computed for a module can be reused. Assuming a previous rewriting of $\mathbf{A}(\overline{v})$ has generated the term $\mathbf{NT}$ and the label l, if $\overline{v} \overset{l}{\rightharpoonup} \overline{w}$ then $\mathbf{NT}$, with some small modifications, can be used as the result of rewriting $\mathbf{A}(\overline{w})$.

Definition 3.2 (Projection Relation)
$\overline{v} \xrightarrow{l} \overline{w}$ if and only if one of the following is true:

- $\overline{v}$ and $\overline{w}$ are empty list,
- $\overline{v} = [l_x : x \mid \overline{v'}]$, $\overline{w} = [\bot : y \mid \overline{w'}]$, $\overline{v'} \xrightarrow{l} \overline{w'}$, and $x = y$ or $l_x \not\subseteq l$.

Let $\mathbf{A}$ be a module name, $\overline{v}$ a list of values, $\boldsymbol{T}$ the term in which the module $\mathbf{A}(\overline{v})$ is rewritten into one step, and l the label generated by this rewriting process. Let $\overline{w}$ be a list of values such that $\overline{v} \xrightarrow{l} \overline{w}$, and $\boldsymbol{S}$ the term in which the module $\mathbf{A}(\overline{w})$ is rewritten into one step. Theorem 3.3 states that $\boldsymbol{T}$ and $\boldsymbol{S}$ have the "same structure".

Theorem 3.3 (One step dependency) *For all module name $\boldsymbol{A}$, values list $\overline{v}$, and set of rewriting rules $\mathcal{R}$, if "$\bot : \boldsymbol{A}(\overline{v}) \rightarrow_{\mathcal{R}} l : \boldsymbol{T}$" then, for all list of values $\overline{w}$ such that $\overline{v} \xrightarrow{l} \overline{w}$, "$\bot : \boldsymbol{A}(\overline{w}) \rightarrow_{\mathcal{R}} \bot : \boldsymbol{S}$" with $\boldsymbol{S} \equiv_S \boldsymbol{T}$.*

Proof (Sketch) Theorem 3.3 relies on the fact that the same rewriting rule is applied in both cases. It comes from the way the generated label is constructed. Indeed, all the parameters influencing the selection of the rewriting rule to apply to this module belong to the generated label. As a consequence, rewriting the same module, with parameters into which the original parameters project themselves, will apply the same rewriting rule. And then, it will return a term with the "same structure". □

Example 3.4 (Application of Theorem 3.3)
Using the FL-system described in example 2.2, one rewriting step on "$l_a : \mathbf{A}(l_1 : 0, l_2 : 2)$" yields the term "$l_a \cup l_1 : \mathtt{t}(2)$". Therefore, Theorem 3.3 implies that, for any list of values $\overline{w}$ such that $[l_1 : 0, l_2 : 2] \xrightarrow{l_a \cup l_1} \overline{w}$ (i.e. of the form $[\bot : 0, \bot : \bot]$), one rewriting step on "$\bot : \mathbf{A}(\overline{w})$" yields a term having the same structure than $\mathtt{t}(2)$ (i.e. a single call to the primitive $\mathtt{t}$ but potentially with a different value as parameter).

Let $\rightarrow_{\mathcal{R}}^{\dagger}$ be the function mapping a term to the normal form obtained by the reflexive transitive closure of $\rightarrow_{\mathcal{R}}$. Corollary 3.5 is the generalization of Theorem 3.3 to $\rightarrow_{\mathcal{R}}^{\dagger}$.

Corollary 3.5 (Multi-step dependency) *For all module name $\boldsymbol{A}$, values list $\overline{v}$, and set of rewriting rules $\mathcal{R}$, if "$\bot : \boldsymbol{A}(\overline{v}) \rightarrow_{\mathcal{R}}^{\dagger} l : \mathsf{NT}$" then, for all list of values $\overline{w}$ such that $\overline{v} \xrightarrow{l} \overline{w}$, "$\bot : \boldsymbol{A}(\overline{w}) \rightarrow_{\mathcal{R}}^{\dagger} \bot : \mathsf{NS}$" with $\mathsf{NS} \equiv_S \mathsf{NT}$.*

Proof (Sketch) The proof goes by induction on the number of rewriting steps. For one rewriting step, this corollary is equivalent to Theorem 3.3. Otherwise, for the same reasons than for Theorem 3.3, the first rewriting step of both rewriting process uses exactly the same rule and gives back two terms ($\boldsymbol{T}$ and $\boldsymbol{S}$) with the exact same structure. If a parameter of a module appearing in $\boldsymbol{T}$ influences the structure of NT then its tag is a subset of l (it comes from the fact that a rewriting step only adds identifiers to the tag of the resulting term). If the tag of a parameter in $\boldsymbol{T}$ is a subset of l, it means that all the parameters (in $\overline{v}$) which have been used to compute its value have the exact same numerical value in $\overline{v}$ and in $\overline{w}$ (because

of $\overline{v} \xrightarrow{l} \overline{w}$). Then, as $\boldsymbol{S}$ has been constructed using the same rule than $\boldsymbol{T}$, this parameter has the same numerical value in $\boldsymbol{T}$ and in $\boldsymbol{S}$. So, it is possible to use the inductive hypothesis on all the modules appearing in $\boldsymbol{T}$ in order to finish the proof. □

Interpretation.

As exposed in the beginning of this section, the goal of this dynamic dependency analysis is to characterize a set of modules (S), as big as possible, which are *structurally joinable* with a given module ($\mathbf{A}(\overline{v})$) which has just been rewritten into the term in normal form "l : **NT**". Corollary 3.5 implies that any module $\mathbf{A}(\overline{w})$, such that $\overline{v} \xrightarrow{l} \overline{w}$, is *structurally joinable* with the module $\mathbf{A}(\overline{v})$. The triplet $(A, \overline{v}, l)$ is then a valid characterization of the set S. Any module, whose name is A and whose parameters $\overline{w}$ are such that $\overline{v} \xrightarrow{l} \overline{w}$, will be rewritten in a term in normal form which has the same structure than **NT**.

This property is useful for the main goal of this paper, which is to develop a cache for the rewriting process of FL-systems. For each module rewritten, Corollary 3.5 enables the cache system to characterize a set, potentially infinite, of modules for which it already knows the structure of the rewritten term in normal form. However, this corollary does not help to determine the values of the parameters of the terminals contained in the term in normal form. This is the subject of the next subsection.

3.2 What about the Parameters?

The work presented above enables the cache system to characterize a set S of modules for which it is able to determine the structure of the term in normal form obtained by rewriting them. For FL-systems, the structure of a term in normal form is a sequence of calls to graphic primitives. For the cache system to be fully functional, it must also determine the values of the parameters of the terminals (or graphic primitives) contained in the term in normal form.

In an earlier version of this work [11] (in French), the method used to determine those values fixes the value of any parameter appearing in any expression during the rewriting process. As a consequence, it reduces the size of the set S, and then the power of the cache system. The only generalization achieved by this method concerns parameters which are just transferred from the original module to the graphic primitives. Even if this method is simple, a prototype for a forest of bushes as described in Example 2.1 and Fig. 2 has shown that the benefit of the early version of the *FL-system Intelligent Cache* (FLIC) can reach 25%.

The method presented in this paper uses symbolic execution [9]. This enables the cache system to express the values of the parameters of the terminals in the term in normal form, for the modules belonging to S, as an expression of the original parameters.

A New Semantics Making Use of Symbolic Execution.

In this part of the work, parameters of the axiom are replaced by a unique identifier. Values are now expressions of identifiers and constants. A list of values (i.e.

a list of expressions) is written $\overline{e}$. The label of a value is then the set containing all the identifiers appearing in it ($\texttt{ident}(e) = \bigcup_{id \in e}\{id\}$). Let $\mathcal{S}em_2$ be the semantics doing the same job as $\mathcal{S}em_1$, but with values which are expressions of identifiers and constant. When rewriting the term "**NT** $\mathbf{A}(\overline{e})$ $\boldsymbol{S}$", $\mathcal{S}em_2$ applies the following rule:

$$\frac{\texttt{first}(\mathcal{R}, \mathbf{A}(\texttt{eval}(\overline{e}, \sigma))) \;=\; \mathbf{A}(\overline{p}) : c_* \wedge c \rightarrow \boldsymbol{T} \qquad l_{\mathrm{c}} = \bigcup_{x \in \texttt{var}(c_* \wedge c)} x[\texttt{ident}(\overline{e})/\overline{p}])}{l : \textbf{NT}\ \mathbf{A}(\overline{e})\ \boldsymbol{S} \quad \rightarrow_{\mathcal{R}} \quad l \cup l_{\mathrm{c}} \ :\ \textbf{NT}\ \boldsymbol{T}[\overline{e}/\overline{p}]\ \boldsymbol{S}}$$

Compared to the semantics $\mathcal{S}em_1$ (Sect. 3.1), there are only a few differences.

(i) To rewrite an axiom $\mathbf{A}(\overline{v})$ with $\overline{v}$ a list of numerical values, the first step is to replace the numerical values used as parameters of $\mathbf{A}$ by some unique identifiers. The result is a module $\mathbf{A}(\overline{e})$ whose values are identifiers. A value store σ is then generated from $\overline{e}$ and $\overline{v}$; it maps the i^{th} element in $\overline{e}$ to the i^{th} element in $\overline{v}$. Rewriting $\mathbf{A}(\overline{e})$ with this value store generates a term in normal form **NT** whose values are expressions. Finally, the term in normal form resulting from $\mathbf{A}(\overline{v})$ is $\texttt{eval}(\textbf{NT}, \sigma)$.

(ii) As values are now expressions, it is required to evaluate those expressions before using the function `first`. σ is a value store which maps the identifiers of the parameters of the axiom with their numerical value. The function `eval` is updated to take this store as a parameter and use it to evaluate the expressions appearing in its first parameter. The value of a constant is the constant itself, and the value of a variable id is $\sigma(id)$. For expressions, the function uses the following rule:

$$\frac{\sigma \;\vdash\; x \;\rightarrow_{\mathrm{e}}\; n_x \qquad \sigma \;\vdash\; y \;\rightarrow_{\mathrm{e}}\; n_y}{\sigma \;\vdash\; (x\ op\ y) \;\rightarrow_{\mathrm{e}}\; f_{op}(n_x, n_y)}$$

(iii) The identifiers added in the label of the resulting term (l_{c}) are mostly computed as in $\mathcal{S}em_1$. The only "difference" is that the function `ident` now collects all the identifiers occurring in an expression, instead of extracting the label of a labeled value (as in $\mathcal{S}em_1$).

(iv) As in $\mathcal{S}em_1$, the module rewritten is replaced by the term given by the rewriting rule. The parameters in this term are replaced by their "value" in the module rewritten. However, with $\mathcal{S}em_1$ the expressions obtained are evaluated, with $\mathcal{S}em_2$ the expressions obtained (which are expressions of identifiers) are not evaluated and are now considered "values".

Properties of the New Semantics.

As in $\mathcal{S}em_1$, the rewriting process of a module $\mathbf{A}(\overline{v})$ returns the characterization of a set of modules S and a term in normal form **NT** (values in this term are expressions of identifiers). To get the final term in normal form with numerical values, the function `eval` is called on **NT** with the value store which maps the identifiers of the parameters of $\mathbf{A}$ to their respective value in $\overline{v}$. In fact, as shown by Corollary 3.8, for any module in S with values $\overline{w}$, the only thing to do to get the

rewritten term is to call the function `eval` on **NT** with the value store which maps the identifiers of the parameters of **A** to their respective value in $\overline{w}$.

Let `zip` be the function labeling the values in the first list with the identifiers in the second (i.e. zipping two lists). For example, $\texttt{zip}([1,2],[x,y])$ is the list $[\{x\} : 1, \{y\} : 2]$. Let $\sigma_{\overline{e},\overline{v}}$ be the store mapping the i^{th} element in $\overline{e}$ to the i^{th} element in $\overline{v}$.

Theorem 3.6 (One step rebuilt) *For all module name* $\boldsymbol{A}$*, numerical values list* $\overline{v}$*, identifiers list* $\overline{e}$ *such that* $\overline{v}$ *and* $\overline{e}$ *have the same size, and set of rewriting rules* $\mathcal{R}$*, if, with the value store* $\sigma_{\overline{e},\overline{v}}$*, "* $\bot : \boldsymbol{A}(\overline{e}) \rightarrow_{\mathcal{R}} l : \boldsymbol{T}$ *" then for all list of numerical values* $\overline{w}$*, such that* $\texttt{zip}(\overline{v},\overline{e}) \overset{l}{\rightharpoonup} \texttt{zip}(\overline{w},\overline{e})$*, rewriting* $\boldsymbol{A}(\overline{e})$ *with the value store* $\sigma_{\overline{e},\overline{w}}$ *yields the term* $\boldsymbol{T}$*.*

Proof (Sketch) Theorem 3.6 comes from the way the generated label is constructed. Indeed, all the parameters influencing the selection of the rewriting rule to apply to a module to be rewritten belong to the generated label. As a consequence, rewriting the same module, with parameters into which the original parameters project themselves, will apply the same rewriting rule and then the same operations on the parameters. As there is no numerical evaluation of the values the two terms obtained are the same. □

Example 3.7 (Application of Theorem 3.6)
Using the FL-system described in example 2.2 with $\mathcal{S}em_2$, one rewriting step on "$l_a : \mathbf{A}(m,n)$" with the value store $[m \mapsto 0, n \mapsto 2]$ yields the term "$l_a \cup \{m\} : \mathtt{t}(n)$". Therefore, Theorem 3.6 implies that, for any list of numerical values $\overline{w}$ such that $[\{m\} : 0, \{n\} : 2] \overset{l_a \cup \{m\}}{\rightharpoonup} \texttt{zip}(\overline{w},[m,n])$ (i.e. of the form $[0,\bot]$), one rewriting step on "$\bot : \mathbf{A}(\overline{e})$" with the value store $\sigma_{[m,n],\overline{w}}$ yields the exact same term $\mathtt{t}(n)$.

Corollary 3.8 (Multi-step rebuilt) *For all module name* $\boldsymbol{A}$*, numerical values list* $\overline{v}$*, identifiers list* $\overline{e}$ *such that* $\overline{v}$ *and* $\overline{e}$ *have the same size, and set of rewriting rules* $\mathcal{R}$*, if, with the value store* $\sigma_{\overline{e},\overline{v}}$*, "* $\bot : \boldsymbol{A}(\overline{v}) \rightarrow^{\dagger}_{\mathcal{R}} l :$ **NT***" then for all list of values* $\overline{w}$*, such that* $\texttt{zip}(\overline{v},\overline{e}) \overset{l}{\rightharpoonup} \texttt{zip}(\overline{w},\overline{e})$*, rewriting* $\boldsymbol{A}(\overline{e})$ *with the value store* $\sigma_{\overline{e},\overline{w}}$ *yields the term in normal form* **NT***.*

Proof (Sketch) This corollary follows directly from Theorem 3.6 by doing an induction on the number of rewriting steps. □

The main difference between Corollary 3.8 and Corollary 3.5 is that for Corollary 3.5 only the structure of the two final term in normal form are identical, whereas for Corollary 3.8 the two final term in normal form are exactly identical.

4 Application to an FL-system's Cache

As expressed before, the work presented in this paper aims at developing an efficient cache mechanism for FL-systems. In order to achieve more efficiency, not only the main rewriting process is cached, but also are all the sub-processes. With the small steps semantics presented earlier only the main rewriting process can be cached. The sub-processes are not explicit. To be able to formalize the cache system, the

semantics $\mathcal{S}em_2$ (Sect. 3.2) is rewritten as a big steps semantics in Fig. 3. Let us call $\mathcal{S}em_3$ this big step semantics. The function $\rightarrow^{\dagger}_{\mathcal{R}}$ is the restriction to a range belonging to the normal form terms of the reflexive transitive closure of $\rightarrow_{\mathcal{R}}$.

$$\sigma \vdash \mathbf{NT} \rightarrow^{\dagger}_{\mathcal{R}} \emptyset : \mathbf{NT}$$

$$\frac{\begin{array}{ll} \texttt{first}(\mathcal{R}, \mathbf{A}(\texttt{eval}(\overline{e}, \sigma))) = \mathbf{A}(\overline{p}) : c_* \wedge c \rightarrow \boldsymbol{T} & l_{\mathrm{c}} = \texttt{var}(c_* \wedge c) \\ (\overline{p} \mapsto \texttt{eval}(\overline{e}, \sigma)) \vdash \boldsymbol{T} \rightarrow^{\dagger}_{\mathcal{R}} l_{\mathrm{t}} : \mathbf{NT} & \sigma \vdash \boldsymbol{R} \rightarrow^{\dagger}_{\mathcal{R}} l_{\mathrm{r}} : \mathbf{NR} \end{array}}{\sigma \vdash \mathbf{NS}\ \mathbf{A}(\overline{e})\ \boldsymbol{R} \rightarrow^{\dagger}_{\mathcal{R}} (l_{\mathrm{c}} \cup l_{\mathrm{t}})[\texttt{ident}(\overline{e})/\overline{p}]^{\cup} \cup l_{\mathrm{r}} : \mathbf{NS}\ \mathbf{NT}[\overline{e}/\overline{p}]\ \mathbf{NR}}$$

Fig. 3. Big step semantics of the rewriting system

This semantics uses an environment variable called σ. As previously, σ is a value store. It maps the variables in the term undergoing the rewriting process to their numerical value. The first rule states that a term already in normal form is rewritten into itself. The label generated by this rewriting step is empty. In the second rule, the left-most module is rewritten first and the selection of the rewriting rule to apply is done as in $\mathcal{S}em_2$. `var` collects the parameter names appearing in its argument. Therefore, the parameters influencing the selection of the rewriting rule are given by l_{c}. This label is expressed using parameters in $\overline{p}$. Before including this label in the label of the final term in normal form, l_{c} has to be translated in a label expressed using parameters appearing in $\overline{e}$. This is done by applying the substitution: $l[\texttt{ident}(\overline{e})/\overline{p}]^{\cup}$. This substitution replaces each parameter name ($\overline{p}_i$) in l by its corresponding set ($\texttt{ident}(\overline{e}_i)$); and finally flattens the set of sets obtained. The term on the right side of the rewriting rule ($\boldsymbol{T}$) is rewritten in normal form in **NT**. $\boldsymbol{T}$ is expressed using the parameters in $\overline{p}$. Its rewriting process has then to use a new value store mapping the parameters in $\overline{p}$ to their numerical value. This is done by mapping the parameter at position i to the numerical value obtained by evaluating the value at position i in $\overline{e}$ using the previous value store (σ). In order to express the parameters of the primitive calls in **NT** in function of the initial parameters, the parameters belonging to $\overline{p}$ in **NT** are replaced by the value (i.e. expression of identifiers) at the same position in $\overline{e}$ (**NT**$[\overline{e}/\overline{p}]$). That rewriting process generates the label l_{t}. This label has to follow the same treatment as l_{c} for the same reason. The term at the right of the left-most module ($\boldsymbol{R}$) is also rewritten in a term in normal form (**NR**). This rewriting process generates the label l_{r}. This label is already expressed using the parameters appearing in $\overline{e}$; so it can be directly included in the label of the final term in normal form. This term is obtained by replacing $\mathbf{A}(\overline{e})$ and $\boldsymbol{R}$ by their respective normal form rewritten term.

This semantics returns exactly the same result than $\mathcal{S}em_2$. However, with this semantics, the complete rewritings of sub-processes are explicit. In Fig. 3, the main term rewritten is $\texttt{eval}(\mathbf{NS}\ \mathbf{A}(\overline{e})\ \boldsymbol{R}, \sigma)$. The rule shows explicitly the final result of the sub-process rewriting the module $\texttt{eval}(\mathbf{A}(\overline{e}), \sigma)$; the resulting term in normal form is $\texttt{eval}(\mathbf{NT}, \sigma)$. It is then possible, during the execution of the main rewriting process, to add in the cache a pair (key, value) linking $\texttt{eval}(\mathbf{A}(\overline{e}), \sigma)$ to $\texttt{eval}(\mathbf{NT}, \sigma)$.

The next subsection exposes the method to generate a generic key mapping some modules structurally joinable with $\texttt{eval}(\mathbf{A}(\overline{e}), \sigma)$ and the associated value.

4.1 Generation and Usage of Keys and Values

Generation of pairs (key ,value) to be inserted into the cache.

First, let us introduce a function used for the generation of the keys. `gen` takes as parameters a list of numerical values ($\overline{n}$), a same length list of parameter names $\overline{p}$ (considered as identifiers), and a label l (i.e. a list of identifiers). It returns a list of numerical values with wild-cards ($\#_1..\#_n$). The element at position i in the result is n_i (the element at position i in $\overline{n}$) if and only if $p_i \subseteq l$, otherwise it is the wild-card $\#_i$. For example, $\texttt{gen}([1,2,3],[x,y,z],\{x,y\})$ returns $[1,2,\#_3]$.

Let us consider the caching of the result $\mathbf{NT}$ of the rewriting of $\mathbf{A}(\overline{e})$ with the value store $\sigma_{\overline{e},\overline{n}}$ where $\overline{e}$ is a list of identifiers. We assume that this rewriting process generated the label l. The key used for caching is a module whose identifier is $\mathbf{A}$ and parameters are $\texttt{gen}(\overline{n},\overline{e},l)$. The value is the term $\mathbf{NT}$ where identifiers appearing in $\overline{e}$ are replaced by $\#_i$ where i is the position of this identifier in $\overline{e}$. Assume that $\overline{\#}$ is the following list of arbitrary length n: $[\#_1,\#_2,\ldots,\#_n]$. The value associated to the key is $\mathbf{NT}[\overline{\#}/\overline{e}]$.

Example 4.1 (Generating a cache entry) Using the FL-system described in example 2.2, rewriting $\mathbf{A}(m,n)$ with the value store $[m \mapsto 0, n \mapsto 2]$ yields the term in normal form $\texttt{t}(n)$ and label $\{m\}$. Therefore the pair added in the cache for this rewriting process is $(\mathbf{A}(0,\#_2),\ \texttt{t}(\#_2))$.

Usage of pairs (key ,value) appearing in the cache.

We first define a concretization relation among lists of elements belonging to $\mathbb{R} \cup \bigcup_{i\in\mathbb{R}} \#_i$. A list $\overline{n}_\text{c}$ is a concretization of an other same length list $\overline{n}_\text{a}$ if and only if any element belonging to $\mathbb{R}$ in $\overline{n}_\text{a}$ is equal to the element at the same position in $\overline{n}_\text{c}$. It can also be seen as the formalization that $\overline{n}_\text{c}$ is an "instance" of $\overline{n}_\text{a}$ (with $\#_i$ denoting variables, as _ in Prolog). It is denoted $\overline{n}_\text{c} \succ \overline{n}_\text{a}$. A formal definition of this relation follows.

Definition 4.2 (Concretization relation)
$\overline{n}_\text{c} \succ \overline{n}_\text{a}$ is true if and only if one of the following is true:

- $\overline{n}_\text{c}$ and $\overline{n}_\text{a}$ are empty list,
- $\overline{n}_\text{c} = [\ x \mid \overline{m}_\text{c}\ ]$, $\overline{n}_\text{a} = [\ y \mid \overline{m}_\text{a}\ ]$, $\overline{m}_\text{c} \succ \overline{m}_\text{a}$, and $x = y$ or $y \notin \mathbb{R}$.

Let $\overline{e}$ be a list of size s of expressions of identifiers, $\overline{i}$ be a list of size s of identifiers and $\overline{n}$ be a list of size s of real numbers. Before starting the rewriting process for the module $\mathbf{A}(\overline{e})$ with the value store $\sigma_{\overline{i},\overline{n}}$, the cache system checks if there exists a pair $(\mathbf{A}(\overline{n}_a), \mathbf{NT})$ such that $\texttt{eval}(\overline{e},\sigma_{\overline{i},\overline{n}}) \succ \overline{n}_a$. If that is the case, then the cache mechanism returns directly the term in normal form $\mathbf{NT}[\texttt{eval}(\overline{e},\sigma_{\overline{i},\overline{n}})/\overline{\#}]$. This term is the result which would have been obtained by executing all the rewriting process for $\mathbf{A}(\texttt{eval}(\overline{e},\sigma_{\overline{i},\overline{n}}))$.

Example 4.3 (Using a cache entry) If the cache contains a pair $(\mathbf{A}(0,\#_2),\ \texttt{t}(\#_2))$, it means that whenever rewriting the module whose name is $\mathbf{A}$ with a first parameter

equal to 0, and whatever value as second parameter, the final term in normal form is the terminal `t` with the value of the second parameter of **A** as argument. For example, the rewriting process of $\mathbf{A}(0, 2)$ should return `t`(2).

5 Conclusion

In this paper, we presented an *intelligent* cache mechanism called *FL-system intelligent cache* (FLIC). This mechanism is based on a dynamic dependency analysis and on symbolic execution. This new cache mechanism for FL-systems enables to take more benefit from the high level of redundancy which appears in relatively large virtual environments. The reuse of geometric objects, which has to be made explicit with standard methods, is partially automatic and implicit with the FLIC. This cache is more efficient than a simple cache due to the higher generality of the pairs (key, value) generated with our method. It is then possible to get from the cache the rewritten form of a module which has never been rewritten. This rewritten form is obtained by instantiation of a value whose key is "sufficiently similar" to the module to rewrite.

To the best of our knowledge, except for [2], there is no equivalent work. The authors of [2] use similar methods to achieve similar goals in the λ-calculus. Their cache maps λ-terms to their normal form value. The main differences with our work are, first, that the context is slightly different; and that the values in our cache are not normal form values, but generalizations of normal form terms. This generalization enables our cache to be more efficient. In computer graphics, there exist few works aiming at similar goals. It is frequent to try to reuse geometric objects already defined somewhere else. The two main motivations for this are to reduce the size of the model and to enhance the efficiency of the rendering. For example, Hart[6] proposed the *instantiation of geometric objects* which is now integrated in the majority of geometric modeling languages (e.g. VRML[13]). In the area of natural environments modeling, Deussen et al. [5] propose the *approximate instantiation* as an extension of Hart's instantiation paradigm. Procedurally generated objects are grouped in clusters in function of the values of their parameters. For each cluster, only one geometric object is generated. This object is used as the approximate instance of all the procedurally generated objects belonging to that cluster. This method reduces the number of objects in the environment (each object can be reused plenty of times in the same environment). However, it brings approximation. This approximation is minimized by assuming, without checking it, that two procedurally generated objects having similar parameters generate similar geometric objects. In our opinion, the FLIC is an optimization complemental to the approximate instantiation. Use of Deussen's technique on the key generated by the cache mechanism would increase the precision of the approximate instantiation and allow the environment generation to benefit from both techniques. In the context of 3D tree computation, Kang et al. [8] propose the creation of a library of random substructure instances randomly assembled at plant level in order to speed up the generation of plant topology without altering the visual realism of the simulated trees. In their work, the plant is explicitly described as an assembly of substruc-

tures, whereas in this paper, reuse of substructures is automatically deduced by the rewriting mechanism.

The cache mechanism proposed in this paper is under integration into a virtual environment generation platform. This platform is already under development and partially functional (Fig. 2 has been generated using this platform). It will be used to test the benefit given by the symbolic evaluation. It will also serve to fine tune some of the features of the cache mechanism; for example, the management of the pairs (key, values) by the cache. Future work will involve exploring the complementarity of the cache mechanism and the approximate instantiation, as well as fine tuning the cache mechanism in order for the platform to take the most benefit from it.

Acknowledgments. The authors are grateful to the reviewers for their insightful and helpful comments.

References

[1] Abadi, M., A. Banerjee, N. Heintze and J. G. Riecke, *A core calculus of dependency*, in: *Proc. Principles of Programming Languages*, 1999, pp. 147–160.

[2] Abadi, M., B. Lampson and J.-J. Lévy, *Analysis and caching of dependencies*, in: *Proc. Functional Programming* (1996), pp. 83–91.

[3] Baader, F. and T. Nipkow, "Term Rewriting and All That," Cambridge University Press, 1998.

[4] Dershowitz, N. and J.-P. Jouannaud, *Rewrite systems*, in: J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, Volume B: **Formal Models and Semantics**, Elsevier Science, Amsterdam, 1990 pp. 243–320.

[5] Deussen, O., P. Hanrahan, B. Lintermann, R. Měch, M. Pharr and P. Prusinkiewicz, *Realistic modeling and rendering of plant ecosystems*, in: *Proc. Computer Graphics and Interactive Techniques* (1998), pp. 275–286.

[6] Hart, J. C., *The object instancing paradigm for linear fractal modeling*, in: *Proc. Graphics Interface* (1992), pp. 224–231.

[7] Hartman, J. and J. Wernecke, "VRML 2.0 Handbook, The: Building Moving Worlds on the Web," Addison Wesley Professional, 1996, 1 edition.

[8] Kang, M., P. De Reffye, J. Barczi, B. Hu and F. Houllier, *Stochastic 3d tree simulation using substructure instancing*, in: *Proceedings of the International symposium on plant growth modeling, simulation, visualization and their applications* (2003), pp. 154–168.

[9] King, J. C., *Symbolic execution and program testing.*, Commun. ACM **19** (1976), pp. 385–394.

[10] Le Guernic, G. and T. Jensen, *Monitoring information flow*, in: A. Sabelfeld, editor, *Proc. Workshop on Foundations of Computer Security* (2005), pp. 19–30, lICS'05 Affiliated Workshop.

[11] Le Guernic, G. and J. Perret, *FL-system's Intelligent Cache*, in: A. Vautier and S. Saget, editors, *Proc. MajecStic*, Rennes, France, 2005, pp. 79–87.

[12] Lindenmayer, A., *Mathematical models for cellular interactions in development, I & II*, Journal of Theoretic Biology **18** (1968), pp. 280–315.

[13] Marrin, C., R. Carey and G. Bell, *A VRML specification*, Technical report, VRML consortium (1997), http://www.vrml.org/Sprecifications/VRML97.

[14] Marvie, J.-E., J. Perret and K. Bouatouch, *The FL-system: A functional L-system for procedural geometric modeling.*, The Visual Computer **21** (2005), pp. 329–339.

[15] Prusinkiewicz, P., A. Lindenmayer, J. S. Hanan et al., "The algorithmic beauty of plants," Springer-Verlag, New York, 1990.

[16] Prusinkiewicz, P., L. Mundermann, R. Karwowski and B. Lane, *The use of positional information in the modeling of plants*, in: *Proc. Computer Graphics and Interactive Techniques* (2001), pp. 289–300.

[17] Pugh, W. and T. Teitelbaum, *Incremental computation via function caching*, in: *Proc. Principles of Programming Languages*, 1989, pp. 315–328.

[18] Shreiner, D., "OpenGL Reference Manual: The Official Reference Document to OpenGL," OpenGL, Addison Wesley Professional, 2004, 4 edition.