Dependent L-System

by Sven Nilsen, 2018

In this paper I formalize the semantics of a Dependent L-System in path semantics. This has potential applications in procedural generation, rendering of computer graphics and constraint solvers.

If you are only interested in potential applications, skip to “Potential Applications”.

An L-System or Lindenmayer-System is a simple parallel rewriting system and a type of formal grammar. It was introduced and developed by Aristid Lindenmayer in the field of biology to describe growth of plant cells.

For example:

$A \rightarrow AB$

A
AB
ABB
ABBB
…

Consider the symbols in an L-System to be types. A Dependent L-System associates a sub-type with each symbol that the rules for rewriting can use to compute new sub-types.

$A \rightarrow AA$

$[\text{size}]\ N \rightarrow [\text{size}]\ N/2,\ [\text{size}]\ N/2$

A rule can be applied under a condition, which is the same as defining a partial rule:

$[\text{size}]\ (N : \text{even} \wedge (> 1)) \rightarrow [\text{size}]\ N/2,\ [\text{size}]\ N/2$

Dependent L-Systems are a superset of Parametric L-Systems.

Rules of Dependent L-Systems are closed under non-commuting Finite Boolean algebra:

$R_2 = R_0 \wedge R_1$	\`$R_2$\` satisfies \`$R_0$\` and \`$R_1$\`
$R_2 = R_0 \vee R_1$	\`$R_2$\` consists of \`$R_0$\` and \`$R_1$\`
$R_0 \vee R_1 \neg= R_1 \vee R_0$	The OR operator does not commute

For example, if you have a rule system using rules \`$R_a$\` and \`$R_b$\` which you want to satisfy \`$R_c$\`, then there exists some rule system of containing rules \`$R_a \wedge R_c$\` and \`$R_b \wedge R_c$\` (in that order):

$(R_a \vee R_b) \wedge R_c = (R_a \wedge R_c) \vee (R_b \wedge R_c)$

Finite Boolean Algebra means that there is no complement operator `¬`. However, one can use relative complement, written as a minus sign `a - b` or with backslash `a \ b`. Rules used for relative complement are interpreted as borrowing their conditions for filtering.

Like functions, a rule in a Dependent L-System has an analogue of an existential path:

$R := A \rightarrow AB$

$\exists R \Leftrightarrow (= AB)$

The existential path has the type (it matches on a list of any type):

$\exists R : [any] \rightarrow bool$

Notice: The interpretation of the existential path of rules is different from that of functions.

The following notation is a short version for type checking:

- `(= A)` means “type is A”
- `(= AB)` means “first object has type A and second object has type B”

For example, `∃R <=> (= ABA)` the length of the list must be 3 for `∃R` to return `true`.

Since any object which there is no rule for is unchanged, the existential path for rules is taken to mean objects that are generated by a rule that are not identical to the input object. This is to simplify algebraic rules of existential paths of combined rules.

Therefore, the identity rule that returns the same object has an existential path that returns `false`:

$R_{id\{T\}} := T \rightarrow [T; 1]$ `T` is here a generic type
$\exists R_{id\{T\}} \Leftrightarrow false_{1\{[any]\}}$

The trivial path of a rule is equivalent to the condition:

$R := A \rightarrow B$
$\forall R \Leftrightarrow (= A)$

The actual generated objects are the union of existential and inverted trivial path as a list of length 1:

$\exists R \vee [\neg \forall R; 1]$

So, with two rules, the actual generated objects are:

$\exists R_0 \vee \exists R_1 \vee [\neg(\forall R_0 \vee \forall R_1); 1]$

Which can also be written:

$\exists R_0 \vee \exists R_1 \vee [\neg \forall R_0 \wedge \neg \forall R_1; 1]$

Assume there is a function `f : A → X` and some function `g` which outputs a value that is matched by some function `h` depending on the output of `f`:

$$\begin{aligned} &R : A \to B \\ &[f]\ x \to [g]\ h(x) \end{aligned}$$

The existential path is then defined by constraining the input of `h` with the existential path of `f`:

$$\exists R <=> (= B) \wedge [g]\ [\exists h\{\exists f\}]\ \text{true}$$

Since `[g] h(x)` is the same as `[g] (= h(x))`, one can remove `[g]` (`h` has type `X → B`):

$$\begin{aligned} &R : A \to B \\ &[f]\ x \to (= h(x)) \\ &\exists R <=> (= B) \wedge [\exists h\{\exists f\}]\ \text{true} \end{aligned}$$

This can also be written in a short version:

$$\begin{aligned} &R : A \to B \\ &[f]\ x \to h(x) \\ &\exists R <=> B \wedge \exists h\{\exists f\} \end{aligned}$$

When there is a condition `x : C`, it is added as a constraint on the input of `h`:

$$\begin{aligned} &R : A \to B \\ &[f]\ (x : C) \to h(x) \\ &\exists R <=> B \wedge \exists h\{\exists f \wedge C\} \end{aligned}$$

When the condition `C : A → bool` at top level, one can do the following:

$$\begin{aligned} &R : A \to B \\ &C \wedge [f]\ x \to h(x) \\ &[f\{C\}]\ x \to h(x) \qquad \text{See paper “Reduction of Proofs with Multiple Constraints”} \\ &\exists R <=> B \wedge \exists h\{\exists f\{C\}\} \end{aligned}$$

This should provide a starting point for a framework for reasoning about Dependent L-systems.

Potential Applications

The advantage of Dependent L-System is flexibility and parallelism. A large class of algorithms related to procedural generation, rendering in computer graphics and some constraint solvers can be formalized using Dependent L-System as a language. By proving that some algorithm is isomorphic to a Dependent L-System, there exists some parallelized version of the algorithm (improving performance).

For example, when rendering computer graphics, it is common to use a scene graph and display lists. In many cases such algorithms can be transformed to some Dependent L-System. By factoring out decisions to rules and types, instead of constructing the scene graph sequentially, one can generate a grammar for a coarse-grained version of the scene graph which is then processed in parallel. Similarly, many procedural generation algorithms might be described as Dependent L-Systems.