Specification and Validation of Algorithms Generating Planar Lehman Words

Alain Giorgetti, Valerio Senni

Specification and Validation of Algorithms Generating Planar Lehman Words

Alain Giorgetti
Inria, Villers-lès-Nancy, F-54600, France, CASSIS project
University of Franche-Comté, FEMTO-ST, UMR 6174, Besançon, F-25030, France
alain.giorgetti@femto-st.fr

Valerio Senni
Department of Computer Science, Systems, and Production
University of Rome Tor Vergata
Via del Politecnico 1, 00133 Roma, Italia
senni@disp.uniroma2.it

Abstract

This paper presents specifications and implementations of algorithms for the generation of planar Lehman words (that is, A. B. Lehman's code for rooted planar maps), together with their validation. The focus is on computer assistance for the task of validation of an implementation with respect to a different implementation or a formal specification. The paper also provides some combinatorial results that are, to our knowledge, new.

1 Introduction

Software quality is higher when the software results from a design method, rather than from program-and-debug cycles. A good method is to write first a formal specification, close to a mathematical description of what the software should do. The specification is written in a high-level language, in order to be compact, easy to write and read, and thus to offer much confidence about its adequacy to the addressed problem. The specification is not intended to provide suggestions on how to solve the problem and is often not executable. If it is executable, it does not target efficiency. The stepwise refinement method consists in transforming the specification into an executable program, while guaranteeing that the program satisfies the specification. When refinement cannot be applied, model-checking, testing or proof methods can be used to check programs against their specification. Our goal is to apply these software engineering methods to the design of generation algorithms. As specification language we use *logic programming*, which is a declarative programming paradigm based on first-order logic and automated theorem proving. Prolog systems such as [3, 7, 9], are implementations of a restricted form of logic programming striving for performance. Prolog is an efficient language and is very well suited for algorithm prototyping due to its closeness to first-order logic specification languages.

The paper applies the testing method to the case study of constructive generation of all the Lehman codes for rooted planar (genus 0) maps, with a prescribed number of symbols, without storing them. We suggest to call them *planar Lehman words*. A *planar Lehman word* (PLW for short) is the shuffle of two Dyck words on two disjoint alphabets (say $\{(,)\}$ and $\{[,]\}$) containing no subword $[(]$ composed of two matching pairs $[]$ and $()$ in the Dyck words, where the four letters in the subword are not necessarily adjacent in the shuffle. A Dyck word on the alphabet $\{a, b\}$ is a word generated by the context-free grammar $S \to \varepsilon \mid aSbS$, where ε is the empty word. Dyck words, or well-parenthesed words, are in bijection with many combinatorial structures such as binary trees, lattice paths, polygon triangulations,

etc. The *length* of a word is its number of symbols. Since the length of Dyck words, shuffles of Dyck words and PLWs is always even, we define their *size* as being half their length, i.e. their number of pairs of symbols.

In 1983, T. Walsh [11] proposed an algorithm to generate rooted maps of any orientable genus without storing them. The algorithm proceeds by generating the words proposed by A. B. Lehman to encode these maps. It can be restricted to the generation of PLWs. Walsh's algorithm constructs a first PLW by concatenation of a Dyck word on parentheses and a Dyck word on brackets. From a generated PLW it generates the next one, whenever possible, by moving parentheses. The algorithm runs in $O(n^2)$ worst-case time, where n is the number of pairs of symbols. Recently, Walsh [12] proposed an extension and an improvement of his algorithm, together with an implementation in C.

The present work explores another way to generate PLWs, by generating each word letter by letter, from left to right, and backtracking as few steps as possible to get one word from the previous one. By Lehman's encoding, this algorithm can also serve to generate all the planar rooted maps with a prescribed number of edges. This case study differs from the rooted planar subcase of [11, 12] in three ways. First, it does not address efficiency in map generation, but only in Lehman word generation. Second, the number of parenthesis pairs and the number of bracket pairs are not distinguished, for sake of simplicity. Third, the focus is more on the design and validation methodology than on the resulting algorithm. The present ongoing work is intended to serve as a methodological guideline for further studies.

The paper is organized as follows. Section 2 proposes a logical specification of PLWs. Section 3 presents a new algorithm to generate PLWs. Section 4 shows its interest for the enumeration of PLWs. Section 5 presents the validation methodology and its results. Section 6 reports about the time efficiency of various generation algorithm implementations. Section 7 contains the conclusion.

2 Specification

We present here first-order Prolog specifications of the Dyck and PLW languages. We will show that those specifications are indeed executable and can be used as generators for those languages.

Prolog is a declarative programming language based on the logic programming paradigm, where programs are sets of first-order clauses and computation is performed by resolution-based theorem proving. First, we briefly recall the logic programming framework; for more details we refer the reader to [4]. In logic programming notation, a comma denotes a conjunction and the symbol `:-` denotes the implication $\leftarrow$. Strings denote variables, if they start with a capital letter, or constants, otherwise. Comments are started by `%`. When variables need not be named, they are replaced by `_`. A Prolog *program* P is a finite set of *clauses* of the form H `:-` $L_1, \ldots, L_n$. where H is an atom called *head* and $L_1, \ldots, L_n$ is a conjunction of literals called *body*. A Prolog program P has a precise denotational semantics (denoted $M(P)$ and called *model*), which is given in terms of a (possibly infinite) set of ground atoms built only using function and predicate symbols occurring in P (the so-called Herbrand models). These atoms are said to *hold* in P and are derivable from P by resolution.

We now define our Prolog-based word generators. For technical reasons, the symbols (,), [and] are replaced by the characters `p`, `a`, `b` and `r`, respectively (standing for `pa`-rentheses and `br`-ackets). Words are encoded by Prolog lists, like `w = [a,b,p,a,r]`. In order to keep track of matching parentheses we will associate a number with each parenthesis (its *label*); so for example a parenthesis `p` is represented as `p(`n`)`, for some $n \geq 0$, and similarly for `a`, `b` and `r`. The above list `w` is indeed of the form `[a(`i`),b(`j`),p(`k`),a(`l`),r(`m`)]`, for some $i, j, k, l, m \geq 0$, and its *abstract* word counterpart α(`w`) is `abpar`. Other encodings of words are possible, but these choices contribute to a cleaner presentation. The grammars $\mathcal{G}_1$: $S \to \varepsilon \mid \mathtt{p}\, S\, \mathtt{a}\, S$ and $\mathcal{G}_2$: $S \to \varepsilon \mid \mathtt{b}\, S\, \mathtt{r}\, S$ which generate the Dyck languages on {`p`,`a`} and {`b`,`r`}, respectively, can be encoded in Prolog as shown in Figure 1 (where clauses are identified by numbers). In particular, the ground atom `dw_pa(w,m,n)` is in $M(\{1, 2\})$ iff the list `w` is of length 2`n` ≥ 0, the parentheses labels start from `m` and the word α(`w`) is generated by $\mathcal{G}_1$. Similarly for the ground atom `dw_br(w,m,n)`, considering the model $M(\{3, 4\})$ and the grammar $\mathcal{G}_2$. Note that matching parentheses take the same integer label and, therefore, there are `n` distinct labels in a word `w` satisfying `dw_pa(w,m,n)`

or `dw_br(w,m,n)`. We assume the availability of the following auxiliary predicates: (1) `in(x,min,max)` holds iff `x` belongs to the interval `[min..max]`, (2) `x is e` holds iff the evaluation of the expression `e` is unified with the variable `x`, and (3) `append(u,v,w)` holds iff $w = u \circ v$, where $\circ$ denotes the list concatenation operator. Note that we parameterize word generation in terms of word length, which is useful for generation purposes but not strictly required for formal definition of the languages.

```
 1. dw_pa([],0,_).                          3. dw_br([],0,_).
 2. dw_pa( W,L,C) :-                        4. dw_br( W,L,C) :-
      in(LU,0,L), LV is L-LU-1,                   in(LU,0,L), LV is L-LU-1,
      LV>=0, D is C+1, E is LU+D,                 LV>=0, D is C+1, E is LU+D,
      dw_pa(U,LU,D), dw_pa(V,LV,E),               dw_br(U,LU,D), dw_br(V,LV,E),
      append([p(C)|U],[a(C)|V],W).                append([b(C)|U],[r(C)|V],W).

 5. shuffle(   [],   [],   []).
 6. shuffle([X|U],   [],[X|U]).
 7. shuffle(   [],[X|V],[X|V]).
 8. shuffle([H|U],[K|V],[H|W]) :- shuffle(U,[K|V],W).
 9. shuffle([H|U],[K|V],[K|W]) :- shuffle([H|U],V,W).

10. dws(W,L) :- in(LU,0,L), LV is L-LU, LV>=0,
                dw_pa(U,LU,0), dw_br(V,LV,0), shuffle(U,V,W).

11. canonical(W) :- \+ noncanonical(W).

12. noncanonical(W) :- append(C,[a(M)|_],W), append(B,[r(N)|_],C),
                       append(A,[p(M)|_],B), append(_,[b(N)|_],A).

13. plw(W,N) :- dws(W,N), canonical(W).
```

Figure 1: Prolog Specification of Dyck Words on {p,a} and {b,r} of length L, and their Shuffles.

As a second step, we define Dyck word shuffles, which are words obtained by arbitrary interleaving of a Dyck word on $\{\mathtt{p},\mathtt{a}\}$ and a Dyck word on $\{\mathtt{b},\mathtt{r}\}$. Shuffles of Prolog lists are easily computed by the predicate `shuffle`, defined by clauses $\{5,6,7,8,9\}$: a shuffle of two lists is built in an inductive way by nondeterministically taking the first element of the first or the second list until either of the two lists is empty. By clause 10, we define shuffles of Dyck words (of length `L`) as shuffles of any two (list-represented) Dyck words `U` (of length `LU`) and `V` (of length `LV`) respectively on $\{\mathtt{p},\mathtt{a}\}$ and $\{\mathtt{b},\mathtt{r}\}$ (for $\mathtt{L} = \mathtt{LU} + \mathtt{LV}$).

For the application we are interested in, nondeterminism is part of the problem. However, we do not want to generate several times the same solution and this is a form of nondeterminism we would like to avoid when writing predicates definitions. In the case of the predicate `shuffle`, for example, the first two arguments are assumed to be ground at evaluation time, so we ensure clause heads are instantiated in an exclusive way, to restrict the nondeterminism (which is unavoidable) only to clauses $8, 9$.

In general, a shuffle of two Dyck words encodes a rooted planar map with a distinguished spanning tree. The property that the word cannot contain the forbidden subword [(]) ensures that the spanning tree is the one obtained by a depth-first search, with the darts incident to each vertex encountered in the cyclic order representing rotation around that vertex according to the orientation imposed upon the sphere. This property was called *canonicity* by Lehman, indicating that the canonical spanning tree is chosen. The last step amounts to specifying in Prolog the characteristic property of canonicity of planar Lehman words. A (list-represented) word `w` is canonical iff there exist no (possibly empty) lists `w1`,..., `w5` and labels `n` and `m` such that `w` can be decomposed into $\mathtt{w1} \circ \mathtt{[b(n)]} \circ \mathtt{w2} \circ \mathtt{[p(m)]} \circ \mathtt{w3} \circ \mathtt{[r(n)]} \circ \mathtt{w4} \circ \mathtt{[a(m)]} \circ \mathtt{w5}$.

By using the `append` predicate and the built-in negation operator `\+` of Prolog we can express canonicity as in clauses 11 and 12. Since clauses are universally quantified, those variables that occur in the body of a clause and not in its head are *existentially quantified*. Note that, among the existential variables in the body of clause 12 the unnamed variables `_` correspond to the lists `w1`,...,`w5` above, while the variables

`A`, `B`, `C` are linking variables. The set of planar Lehman words is defined by clause `13` as the set of Dyck word shuffles of length `n` that satisfy canonicity.

As we pointed out at the beginning of this section, this declarative specification of planar Lehman words is indeed executable. Let us now illustrate how the resolution-based proof procedure of Prolog is used as a computation mechanism. A Prolog system answers to a user query $Q = Q_1, \ldots, Q_k$ against a program P, where $Q_1, \ldots, Q_k$ is a conjunction of (possibly non-ground) literals. An *answer* to $Q_1, \ldots, Q_k$ is a substitution ϑ such that $\forall\ (Q_1, \ldots, Q_k)\vartheta$ is a logical consequence of P; so every ground instance of $Q_i\vartheta$ belongs to $M(P)$. The proof strategy tries, by resolution, to prove (in a left-to-right order) that each atom in the query holds and, at each resolution step, rewrites the current goal into a new one, according to proof rules that are expressed by the program clauses. A clause of the form $H :- A_1, \ldots, A_n.$ indicates that (any instance of) the atom H holds if (the corresponding instance of) the goal $A_1, \ldots, A_n$ hold. Resolution proceeds by replacing (an instance of) the atom H with (the corresponding instance of) the goal $A_1, \ldots, A_n$ and producing a substitution ρ. Clauses with empty body contribute to termination of the proof process, which ends when the current goal becomes empty. A sequence of resolution steps terminating with an empty goal is called an SLD-*refutation*, and has an associated substitution ϑ which is the composition of the substitutions computed by each resolution step. Nondeterminism arises naturally since, at each step in the proof, many clause heads may be used to enact a resolution step. All possible answers can be computed by backtracking to choice-points and choosing a different clause for the specific resolution step.

In Prolog, negation is a first-class programming concept and it is dealt with by considering only *ground* queries (so one has to ensure that an atom under negation has been fully instantiated before its call). In particular, `\+`A holds iff A has no SLD-refutation.

We consider the evaluation of the running example of this section: clause `13` entails a generate-and-test behavior which first computes a Dyck word shuffle of length n and then checks for canonicity. By backtracking, the program generates all PLWs, but this is done by generating *each* Dyck word shuffle of the given length and succeeding only for those that satisfy canonicity. Note that, whenever we fix a value n for the length of the words, both the set of Dyck words and of PLWs is finite and the generation of all of the possible answers terminates. This is clearly not an efficient approach to this problem (see timings in Table 1 and related discussion in Section 6). However, one can be easily convinced of the correctness of the algorithm given in Figure 1.

The advantage of having an executable specification is the possibility to validate more sophisticated algorithms by comparison with the specification we have provided. In particular, in Section 3, we introduce a clever generator for PLWs and we are able to validate it on several instances of the input n.

3 Automata-Based Generation

The language of Dyck words is algebraic. It can be recognized by a pushdown automaton. More precisely, it can be recognized by a one-counter automaton, in the sense of Minsky [5], i.e. a pushdown automaton with only one stack symbol. More generally, the language of shuffles of k ($k \geq 1$) Dyck words can be recognized by an automaton with n counters, whose transitions can only increment or decrement non-negative counters. We propose here an automaton recognizing the language of planar Lehman words, but with an unbounded number of counters, stored in a stack.

For generation purposes, we focus on deterministic automata. They work as acceptance machines by reading a word from left to right, letter by letter, until reaching an acceptance state. But these deterministic automata can also work as generators of the languages they accept, by an exhaustive exploration of all the paths from their initial to their acceptance states, during which the accepted letters are concatenated to form a generated word. We call *automata-based generation* this way of generating families of words (i.e. languages). Such a generation may be efficient if the exploration avoids failure as much as possible. This section presents an automata-based generation of planar Lehman words and its Prolog implementation.

We first explain the reasoning that led us from the definition of planar Lehman words (as shuffles of

two Dyck words with forbidden subwords) to this algorithm. Shuffles of Dyck words on the two disjoint alphabets $\{(,)\}$ and $\{[,]\}$ can be recognized by an automaton using either a stack of symbols or a counter for each alphabet. Assuming that the word w is such a shuffle, the negative property that w does not contain any subword $[(])$ composed of two matching pairs $[]$ and $()$ in the Dyck words is equivalent to the following (positive) property, denoted by C in all that follows: "When reading the word w from left to right, any parenthesis (at height i, opened after the opening of a square bracket [at height j, should be closed by the next parenthesis) at the same height i, before the nearest closure of the square bracket by a square bracket] at height j."

The interest of Property C is that it can be checked during a single pass in the word from left to right, provided the opening symbols [and (are stacked during this pass. When a closing parenthesis) is encountered the topmost opening parenthesis (should be removed from this stack. However searching for this topmost opening parenthesis (in the stack of a pushdown automaton can entail more than one transition, i.e. there are ε-transitions that read nothing. To avoid this, we replace this stack of symbols with a stack of natural numbers, as follows.

In the text a stack of symbols is represented by a word whose first (leftmost) letter is the stack top. A stack of natural numbers is represented by a sequence whose leftmost element is again the stack top. Both are implemented in Prolog by lists in the same order. The reason for this choice will appear clear in Section 4. For $k \geq 1$, the stack of symbols $[^{n_k}$ ($[^{n_{k-1}}$ (... $[^{n_1}$ is represented by the stack of natural numbers n_k, n_{k-1}, ..., n_1. In other words, n_i is the number of opening square brackets [before the i-th opening parenthesis (if $1 \leq i \leq k-1$, and n_k is the number of opening square brackets [after the $(k-1)$-th opening parenthesis (. For instance, the stack of symbols [[[([[[([([[is represented by the stack of counter $3, 2, 1, 0, 0, 2$. The empty stack of symbols is represented by the stack of counters 0.

Figure 2 defines two Prolog predicates `gl0w` that implement the backtracking algorithm explained above. The ground atom `gl0w(w,s,n)` is in $M(\{1, \ldots, 5\})$ iff the list `w` is of length `n` ≥ 0 and the symbols in `w` are closing the pending symbols counted in the stack of natural numbers `s`, whilst satisfying the canonicity property. More precisely, clause `2` encodes the automaton transition which pushes a `0` at the top of the stack `I` when an opening parenthesis (is recognized. Clause `3` recognizes an opening bracket [by incrementing the stack top. When the next symbol if a closing bracket] and the stack top is not null, clause `4` decrements it. Clause `5` specifies that a closing parenthesis) can only be accepted if there are at least two numbers in the stack. Then, the first two numbers at the top of the stack are replaced by their sum, to represent that this closing parenthesis matches the nearest opening one. Finally, clause `6` defines a predicate `gl0w` with two arguments, whose second argument is the number of pairs of symbols.

```
1. gl0w(   [],      [0],0).
2. gl0w([p|W],        I,L) :- L > 0, NewL is L-1,                     gl0w(W,   [0|I],NewL).
3. gl0w([b|W],    [N|I],L) :- L > 0, NewL is L-1, NewN is N+1,        gl0w(W,[NewN|I],NewL).
4. gl0w([r|W],    [N|I],L) :- L > 0, N > 0, NewL is L-1, NewN is N-1, gl0w(W,[NewN|I],NewL).
5. gl0w([a|W],[N1,N2|I],L) :- L > 0, NewL is L-1, NewN is N1+N2,      gl0w(W,[NewN|I],NewL).

6. gl0w(W,Size) :- T is 2*Size, gl0w(W,[0],T).
```

Figure 2: Prolog Predicate for the Generation of PLWs With a Stack of Counters

Profiling the predicate `gl0w` shows that many exploration branches fail producing a word. Intuitively failure happens when the stack counts more pending symbols than authorized to be added to complete a word. Many of these cases can be pruned from the exploration tree by calling `gl0w(w,s,n)` only when `n` is not smaller than the number of pending symbols, what we call here the *control property*. The number of pending symbols can be retrieved from the stack `s`. It is the sum of the counters in the stack `s` plus the stack length minus one. Instead of computing this number several times, it is more efficient to maintain the numbers of pending brackets and parentheses in separate counters. We developed another predicate, named `cl0w` and shown in Figure 3, which checks the control property before each recursive call. It has two more parameters than `gl0w`, respectively counting the numbers of pending brackets and parentheses.

```
1. cl0w(   [],_,_,      [0],0).
2. cl0w([p|W],B,P,        I,L) :- L>0, L>=B+P, NL is L-1, NP is P+1,
                                                          cl0w(W, B,NP, [0|I],NL).
3. cl0w([b|W],B,P,    [C|I],L) :- L>0, L>=B+P, NL is L-1, NC is C+1, NB is B+1,
                                                          cl0w(W,NB, P,[NC|I],NL).
4. cl0w([r|W],B,P,    [C|I],L) :- L>0, L>=B+P, N>0, NL is L-1, NC is C-1, NB is B-1,
                                                          cl0w(W,NB, P,[NC|I],NL).
5. cl0w([a|W],B,P,[C1,C2|I],L) :- L>0, L>=B+P, NL is L-1, NC is C1+C2, NP is P-1,
                                                          cl0w(W, B,NP,[NC|I],NL).

6. cl0w(W,Size) :- T is 2*Size, cl0w(W,0,0,[0],T).
```

Figure 3: Prolog Predicate for the Generation of PLWs, Optimized Using the Control Property

4 Counting

For any non-empty stack s (i.e. sequence) of natural numbers, let $n_{[s],l}$ denote the number of words of length l which are accepted by the automaton defined in Section 3, when starting the recognition from the state described by the stack s. Recall that the ith element in the stack s from right to left is the length of the $(i-1)$th sequence of pending opening brackets in the word under recognition. With this definition the number of planar Lehman words of size e is $n_{[0],2e}$. From the generation algorithm in Section 3, we can derive the following recurrence relation for $n_{[s],l}$.

Proposition 1. *The number $n_{[n_k,\ldots,n_1],l}$ of planar Lehman words which are the concatenation of some word whose sequence of pending symbols is $[^{n_k}$ ($\ldots$ $[^{n_1}$ with some word of length l is defined by*

$$n_{[0],0} = 1, \qquad n_{[s],0} = 0 \text{ if } s \neq 0, \qquad n_{[f,t],l} = n_{[0,f,t],l-1} + n_{[f+1,t],l-1} + r_{[f,t],l-1} + a_{[f,t],l-1} \text{ if } l \geq 1,$$

with $r_{[0,t],l} = 0$, $r_{[f,t],l} = n_{[f-1,t],l}$ if $f \geq 1$, $a_{[f],l} = 0$ and $a_{[f_1,f_2,t],l} = n_{[f_1+f_2,t],l}$, where f, f_1 and f_2 are natural numbers, t is any (possibly empty) sequence of natural numbers, and s is any non-empty sequence of natural numbers.

For $l, r \geq 0$, let $N_{l,r}$ be the (ordinary) generating function defined by

$$N_{l,r}(u_1, \ldots, u_{r+1}) \quad = \quad \sum_{|s|=r+1} n_{[s],l}\ u_1^{n_1}\ \ldots\ u_{r+1}^{n_{r+1}} \tag{1}$$

where s abbreviates the sequence $n_1, \ldots, n_{r+1}$ and $|s|$ denotes its length. By standard techniques in enumerative combinatorics, the former recurrence relation leads to the following proposition, where $\delta_\varphi = 1$ if φ holds, and 0 otherwise.

Proposition 2. *For $l, r \geq 0$ the generating functions $N_{l,r}(u_1, \ldots, u_{r+1})$ are defined by recurrence on l:*

$$\begin{aligned}
N_{0,0}(u_1) &= 1, \\
N_{0,r}(u_1, \ldots, u_{r+1}) &= 0 \text{ if } r \geq 1 \text{ and} \\
N_{l,r}(u_1, \ldots, u_{r+1}) &= N_{l-1,r+1}(0, u_1, \ldots, u_{r+1}) \\
&\quad + u_1^{-1}\,(N_{l-1,r}(u_1, \ldots, u_{r+1}) - N_{l-1,r}(0, u_2, \ldots, u_{r+1})) \\
&\quad + u_1\ N_{l-1,r}(u_1, \ldots, u_{r+1}) \\
&\quad + \delta_{r\geq 1}\,(u_2 - u_1)^{-1} \begin{pmatrix} u_2\ N_{l-1,r-1}(u_2, \ldots, u_{r+1}) \\ -u_1\ N_{l-1,r-1}(u_1, u_3, \ldots, u_{r+1}) \end{pmatrix} \\
&\quad \text{if } l \geq 1.
\end{aligned}$$

We do not provide here a proof of this proposition, but a validation in Section 5 by computation of the first values.

The generating function we are interested in is the summation of $N_{l,r}(u_1, \ldots, u_{r+1})$ for any non-negative values of l and r. Since it would be a generating series with infinitely many indeterminates u_i, we make them implicit in the following definition of the generating function

$$N(x,t) = \sum_{l,r \geq 0} N_{l,r}(u_1, \ldots, u_{r+1}) x^l t^r. \tag{2}$$

Then a notation inspired by explicit substitutions is used to specialize this generating function. In the following proposition, the expression $e_{|y:=0, \forall i \geq p. u_i := t_i}$ represents the expression e where y is replaced with 0 and each indeterminate u_i (for $i \geq p$) is simultaneously replaced with the indeterminate t_i.

Proposition 3. *The generating function $N(x,t)$ is the unique formal power series solution of the equation*

$$N(x,t) \quad = \quad 1 + x \left[\begin{array}{l} t^{-1} \ (N(x,t) - N(x,0))_{|u_1:=0, \forall i \geq 2. u_i := u_{i-1}} \\ + \ u_1^{-1} \ \left(N(x,t) - N(x,t)_{|u_1:=0} \right) + \ u_1 \ N(x,t) \\ + \ t \ (u_2 - u_1)^{-1} \left(u_2 \ N(x,t)_{|\forall i \geq 1. u_i := u_{i+1}} - u_1 \ N(x,t)_{|\forall i \geq 2. u_i := u_{i+1}} \right) \end{array} \right]. \tag{3}$$

Let $L(x)$ be the formal power series in x whose coefficient of x^n is the number of PLWs of length $n \geq 0$. Then $L(x) = N(x,0)$ and Proposition 3 defines it from the bivariate power series $N(x,t)$. Despite its similarity with functional equations arising in lattice path theory, Eq. (3) is not obvious to solve. The question whether it can be solved by the kernel method [1] is open.

Since PLWs of length $2e$ encode rooted planar maps with e edges, it is already known that $L(x)$ is quadratic. This fact reduces the interest of solving Eq. (3), which is presented here because it is to our knowledge a new formula to count PLWs (and rooted planar maps), and a direct consequence of the generation algorithm in Section 3.

5 Validation

Validating programs against the expected behavior can be a fairly complex and time-consuming task, which can greatly benefit of machine support. Most often, complete and formal validation is hard to obtain, while empirical validation can be sufficient to achieve confidence in the correct behavior of the program. Without relying on formal methods or hand proofs, we show how to validate our generation programs by taking advantage of the Prolog language features.

Validity of the program in Figure 1 (which we denote as *specification* program S) is straightforward. On the contrary, validity of the program in Figure 2 (which we denote as *linear* program L) requires more involved arguments. To perform validation we adopt the so-called bounded-exhaustive testing technique [2], in which all the finitely many inputs up to the given size are used to exercise a program. The specification program S is used in two ways: (1) as a *generator*, to enumerate words which must be accepted by L, and (2) as an *acceptor* for words generated by L. If, for a given bound n on word size, each word generated by S is accepted by L and each word generated by L is accepted by S, then S is proven sound and complete, respectively, up to that bound. It is often sufficient to check for small instances of the bound on the input domain in order to reveal bugs in programs.

The advantage of adopting the Prolog language is that symmetric bounded-exhaustive testing can be performed by using a simple query scheme, for a given size `s` and word generators `p` and `q`, as follows:

Q: `(p(W,s), \+ q(W,s)) ; (q(W,s), \+ p(W,s))`

where `;` denotes logic disjunction and `\+` denotes negation (as failure). Prolog evaluation mechanism attempts to satisfy the query Q by instantiating `W` to a word `w` of size `s` such that `p(w,s)` holds and `q(w,s)` does not hold or the converse. If no such word exist then the query fails and the two predicates are equivalent, for size `s`. We should point out that, in our application, this search process terminates since for any given input size the set of generated words is finite. This is not true for general Prolog programs.

Another useful query scheme for validation is the following, for a given size `s` and word generator `p`:

Q': `p(W,s), write(W), fail.`

which allows the enumeration of all the generated words. The query forces the construction of a word `w` of size `s` generated by `p`, its output on a stream, and the failure of the proof mechanism by using the built-in `fail`. Since the proof fails, the backtracking mechanism recovers the last choice-point (necessarily in `p`) and triggers the generation of a new word, until there are no more choice-points.

In order to measure the time taken for answering a query (such as the time for generating the entire word set for a given size) it is possible to use the following query scheme:

Q'': `statistics(runtime,[T1,_]), Goal, statistics(runtime,[T2,_]), Time is T2-T1.`

where the built-in `statistics` is used to measure the CPU time before and after `Goal` execution.

In order to make the validation tasks easier to be performed, we have implemented a validation library [6] that collects several query schemes that are useful for validation purposes. The library provides full automation for symmetric bounded-exhaustive comparison for increasing bound values. It returns counterexamples whenever validation fails (so the debugging process is guided by those counterexamples), and it collects statistics such as generation time and memory consumption. We used our library to validate (up to size $n = 7$) and benchmark our Prolog programs. A package containing all the files needed to run the benchmarks plus some instructions can be found in the library repository [6].

T. Walsh [12] uses Lehman's code to generate maps of any orientable genus up to orientation-preserving isomorphism, and up to a generalized isomorphism that could be orientation-preserving or orientation-reversing. His algorithm proceeds by rejection from rooted maps. PLW generation is the restriction of this general algorithm to genus 0 and rooted maps. T. Walsh kindly provided us his C program implementing his algorithm, named `unsensed.c`, and authorized us to modify it. We thus specialized it into a generation algorithm of PLWs named `rpm.c`. We also translated our predicate `cl0w` into a C program named `cl0w.c`. This C code is available upon request from the authors.

Before comparing the efficiency of these two C programs in Section 6, we discuss here the preliminary task of the validation of Prolog programs against C programs. Our choice has been to store the programs' outputs into files and to compare these files as multisets of terms. This is relevant because words may be generated in a different order and some may be generated several times. Our validation tests, however, show that words are never repeated. All the programs display the PLWs they generate on the standard output using the same syntax.[1] Then, the validation library provides some functionalities to load files contents as Prolog terms and compare them, in a similar way to what is done for Prolog-to-Prolog comparison. We have used this feature to validate the C programs up to size $n = 7$.

Another concern is efficiency of Prolog programs evaluation, which can be comparable neither to that of math-oriented software nor to that of compiled programming languages such as C. However, in Section 6, dedicated to the evaluation of program performances, we show that Prolog evaluation is still manageable for small instances and can be used as a guideline, during algorithm prototyping, for early error detection.

We have not provided proofs to show how we obtained equations of Propositions 1–3 in Section 4, concerning numbers and series of PLWs, from the linear program L. Indeed, apart from lack of space considerations, there is a main drawback on providing such proofs, since the program L itself has not been formally proved equivalent to the (specification) program S. We decided that direct computation of sufficiently many first numbers of the sequences of interest would have provided more confidence in those results. Therefore, we have implemented all the formulas using the Maple computer algebra system and compared the results with those reported in the A000168 series of the OEIS database [8].

We have validated the recurrence relations in Propositions 1 and 2 up to $n = 6$, respectively for all the numbers $n_{[s],2n}$ and all the series $N_{2n,0}$, by executing their recursive implementation in Maple. For Proposition 3, we have extended Maple with a couple of functions supporting explicit substitutions that are lists composed of atomic substitutions $y := a$ and quantified substitutions $(\forall i \geq p.\ u_i := u_{i+b})$ for any variables y and any constants a, p and b. Using these functions, Formula 3 has been validated by truncated expansion in x of $N(x,t)$ up to exponent 12 (i.e. again up to size 6). After specialization of the result to $t = 0$ and $u_1 = 0$, we obtain the series $1 + 2x^2 + 9x^4 + 54x^6 + 378x^8 + 2916x^{10} + 24057x^{12}$ as expected.

[1] Some `printf` statements of the C program written by T. Walsh have been modified for this purpose.

6 Performances

We now experimentally compare the efficiency in time of Prolog and C implementations of various algorithms generating planar Lehman words by increasing size, from size 1 to n for some positive integer n, without storing them. All the programs are executed with a personal computer running Ubuntu with Linux kernel 2.6.24, an Intel Core 2 Duo CPU at 2.66 GHz, and 4 Gb of memory. Prolog programs are executed with SICStus Prolog 3.12.8 [7]. C programs are compiled with GCC 4.4.3.

Generation times are reported (in seconds) in Table 1. Word sizes (numbers of pairs of symbols) are displayed in Column 1. Column 2 displays the number $l_0(n)$ of planar Lehman words of size n generated by these programs. Columns 3 and 4 respectively show the generation times obtained with the Prolog specification `plw.pl` described in Section 2 and its optimization `cl0w.pl` described in Section 3. Column 4 shows the multiplicative factor between these two times. NS stands for "not significant". The other columns compare the C programs `rpm.c` and `cl0w.c`. Columns 6 and 7 show their computation times (in seconds) up to size 10. Column 8 gives the multiplicative factor between these two times.

These results can be interpreted as follows. The specification is many times slower than its stack-based optimization. The decreasing of the reduction rate between both also justifies the optimization effort performed in `cl0w.pl`. However it is worth noticing that the executable Prolog specification already provides more than 200,000 words in five seconds, for a minimal coding effort. This can be sufficient for many applications, in particular to check more intricate algorithms. Our stack-based C program is more efficient than Walsh's program, but the comparison is not completely fair, because the program `rpm.c` additionally groups the words by number of parenthesis pairs. This subject is beyond the scope of the present study, but the first results are promising.

		`plw.pl`	`cl0w.pl`		`rpm.c`	`cl0w.c`	
n	$l_0(n)$	time (s)	time (s)	Factor	time (s)	time (s)	Factor
0	1	0	0	NS	0	0	NS
1	2	0	0	NS	0	0	NS
2	9	0	0	NS	0	0	NS
3	54	0	0	NS	0	0	NS
4	378	0	0	NS	0	0	NS
5	2,916	0.03	0.01	0.33	0	0	NS
6	24,057	0.41	0.08	0.19	0	0	NS
7	208,494	5.13	0.65	0.13	0	0	NS
8	1,876,446	68.95	5.99	0.09	0	0	NS
9	17,399,772	923.94	55.20	0.06	9	1	0.11
10	165,297,834	14,254.97	574.33	0.04	95	12	0.13
11	1,602,117,468	too long	5,226.30	-	1005	125	0.12

Table 1: Generation Times with Prolog (`.pl`) and C (`.c`) Programs

7 Conclusion

We have shown how to use logic programming and bounded-exhaustive testing to design and validate algorithms generating a family of combinatorial objects. This case study shows that declarativeness and the presence of negation and nondeterminism as first-class programming concepts make Prolog an effective tool for prototyping and validating generation algorithms. Bounded-exhaustive testing up to size 6 provides a reasonable confidence in the program correctness.

A useful mechanism for handling formal languages in Prolog is provided by Definite Clause Grammars (DCG), that have been developed to specify context-free grammars but can be easily extended to handle also more expressive grammars. For the sake of simplicity of presentation, we have deliberately avoided

the use of DCGs since they are based on a (more sophisticated) data structure called difference-list (DL) rather than simple lists. Using DLs allows to perform constant-time, rather than linear-time, list append. It should be noted, however, that this brings an advantage only in the specification program S (at the cost of a less clear encoding) while it provides no advantage in the linear program L.

This is still ongoing work, with numerous perspectives. Our priority will be to provide either more computer assistance in the derivation of fast algorithms from their specifications, or formal proofs of their equivalence. Another direction will be to apply the same methodology to other families of Lehman words, i.e. revisit parts of [10] from the viewpoint of software engineering and formal methods. For planar Lehman words, we have many ideas for more efficient generation algorithms than the first two proposed here. We also have a framework ready to validate them.

On the combinatorial side, it is worth noting that planar Lehman words also have an interpretation as lattice paths. They are restrictions of the interpretation of shuffles of two Dyck words as excursions in the bi-dimensional first quarter plane. The step-by-step construction of these paths would lead to the same generation algorithm and counting formulas as proposed here. We found the interpretation as words with forbidden subwords more intuitive.

8 Acknowledgment

We would like to thank Prof. T. R. S. Walsh who carefully read a preliminary version of the present text and made many useful comments and suggestions. We also acknowledge the anonymous referees for their constructive comments.

References

[1] Cyril Banderier and Philippe Flajolet. Basic analytic combinatorics of directed lattice paths. *Theor. Comput. Sci.*, 281(1-2):37–80, 2002.

[2] David Coppit, Jinlin Yang, Sarfraz Khurshid, Wei Le, and Kevin J. Sullivan. Software assurance by bounded exhaustive testing. *IEEE Trans. Software Eng.*, 31(4):328–339, 2005.

[3] GNU. Prolog. `http://www.gprolog.org/`.

[4] John W. Lloyd. *Foundations of Logic Programming, 2nd Edition*. Springer, 1987.

[5] Marvin L. Minsky. *Computation: finite and infinite machines*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1967.

[6] Valerio Senni. Validation library. `https://subversion.assembla.com/svn/validation/`.

[7] SICStus. Prolog. `http://www.sics.se/sicstus/`.

[8] Neil J. A. Sloane. The On-Line Encyclopedia of Integer Sequences, 2012. Sequence A000168, published electronically at http://oeis.org/A000168.

[9] SWI. Prolog. `http://www.swi-prolog.org/`.

[10] Timothy R. S. Walsh. *Combinatorial enumeration of non-planar maps*. PhD thesis, University of Toronto, 1971.

[11] Timothy R. S. Walsh. Generating nonisomorphic maps without storing them. *SIAM J. Alg. Disc. Meth.*, 4:161–178, 1983.

[12] Timothy R. S. Walsh. Generating nonisomorphic maps and hypermaps without storing them. Communication at GASCom'12, 2012.