Specification and Validation of Algorithms Generating Planar Lehman Words

Alain Giorgetti    Valerio Senni

University of Franche-Comté    University of Rome “Tor Vergata”
FEMTO-ST    Dept. of Computer Science,
Inria, CASSIS project    Systems and Production

GASCom 2012
D. Zeilberger: “Teach your computer how to do research!”
D. Zeilberger: “Teach your computer how to do research!”

- Computer-assisted design and validation of efficient enumeration algorithms
D. Zeilberger: “Teach your computer how to do research!”

- Computer-assisted design and validation of efficient enumeration algorithms
  - From a high-level (formal) specification
"D. Zeilberger: “Teach your computer how to do research!”

- Computer-assisted design and validation of efficient enumeration algorithms
  - From a high-level (formal) specification
  - Whenever possible, optimization by transformations
D. Zeilberger: “Teach your computer how to do research!”

- Computer-assisted design and validation of efficient enumeration algorithms
  - From a high-level (formal) specification
  - Whenever possible, optimization by transformations
  - Otherwise, validation of efficient algorithms wrt their specification
    - Intensive testing
D. Zeilberger: “Teach your computer how to do research!”

- Computer-assisted design and validation of efficient enumeration algorithms
  - From a high-level (formal) specification
  - Whenever possible, optimization by transformations
  - Otherwise, validation of efficient algorithms wrt their specification
    - Intensive testing for early error detection
    - Computer proof of equivalence (formal reasoning)
D. Zeilberger: “Teach your computer how to do research!”

- Computer-assisted design and validation of efficient enumeration algorithms
  - From a high-level (formal) specification
  - Whenever possible, optimization by transformations
  - Otherwise, validation of efficient algorithms wrt their specification
    - Intensive testing for early error detection
    - Computer proof of equivalence (formal reasoning)

- Application to Combinatorics
Planar Lehman-Lenormand words

- Code for rooted planar maps
- Proposed by A. Lehman
- Independently proposed by C. Lenormand [Cori75]
- Studied by T. R. S. Walsh
- Shuffles of two Dyck words with forbidden subwords
A Dyck word of length $2n$ (size $n$) contains $n$ pairs of parentheses (possibly nested) which correctly match.
A Dyck word of length $2n$ (size $n$) contains $n$ pairs of parentheses (possibly nested) which correctly match.

Example: $( ( ) ) ( ( ( ) ( ) ) )$
Dyck words
Definition

- A Dyck word of length $2n$ (size $n$) contains $n$ pairs of parentheses (possibly nested) which correctly match.
- Example: ( ( ) ) ( ( ( ) ( ) ) )
- Grammar: $D ::= \varepsilon | ( \ D \ ) \ D$
Planar Lehman words

Definition

**Planar Lehman word**

A planar Lehman word (PLW) is any shuffle of a Dyck word on the alphabet \{ (, )\} and a Dyck word on the alphabet \{ [, ]\}, which does not contain any subword \[ (\) \] composed of two pairs \[\] and ( ) matching in the Dyck words (canonicity property).

- Forbidden pattern \[ . . . ( . . . ) . . . \] . . .
- 9 planar Lehman words with 4 letters:
  
  ( ( ) ) ( [ ] ) ( [ ] ) ( ) ( ) ( ) [ ] [ ] [ ]

- 1 noncanonical Dyck word shuffle with 4 letters: [ ( ) ]
Logic programming

- Programs are sets of rules (Horn clauses) of the form
  \[ H :− A_1 ∧ \ldots ∧ A_n \]
  (meaning, \( H \) holds if \( A_i \) holds for \( i = 1, \ldots, n \))
Logic programming

- Programs are sets of rules (Horn clauses) of the form
  \[ H : - A_1 \land \ldots \land A_n \]
  (meaning, \( H \) holds if \( A_i \) holds for \( i = 1, \ldots, n \))

- Example
  \[
  \begin{align*}
  \text{ordered}([{}]). \\
  \text{ordered}([X]). \\
  \text{ordered}([X_1, X_2 | L]) : - X_1 \leq X_2 \land \text{ordered}([X_2 | L]).
  \end{align*}
  \]
Logic programming

- Programs are sets of rules (Horn clauses) of the form
  \[ H :\neg \ A_1 \land \ldots \land \ A_n \]
  (meaning, \( H \) holds if \( A_i \) holds for \( i = 1, \ldots, n \))

- Example
  
  \[
  \begin{align*}
  \text{ordered([]).} \\
  \text{ordered([X]).} \\
  \text{ordered([X1,X2|L]) :- X1 \leq X2 \land \text{ordered([X2|L]).}}
  \end{align*}
  \]

- Query evaluation
  
  1. Pick leftmost atom in current query: \( Q = A \land R \)
  2. Find unifying head: \( A \sigma = H \sigma \)
  3. Rewrite to get a new query: \( (A_1 \land \ldots \land A_n \land R) \sigma \)
ordered([],).
ordered([X]).
ordered([X1, X2 | L]) :- X1 ≤ X2 ∧ ordered([X2 | L]).

as a generator:

ordered(L).

L = [ ]

L = [X]

L = [X1, X2] with X1 ≤ X2

... L = [X1, X2, X3] with X1 ≤ X2 ∧ X2 ≤ X3
Specification
Labelled Dyck words on \{ (, ) \}

Labels identify matching pairs in words:

\(
\)

Letters encode parentheses and brackets:

'(' → p
')' → a
'[' → b
']' → r

Grammar:

\[
D ::= \varepsilon | (D)D
\]

\(\text{dw}_{pa}(W, L, C)\):

- \(\text{in}(LU, 0, L)\), \(LV > 0\), \(D = C + 1\), \(E = LU + D\),
- \(\text{dw}_{pa}(U, LU, D), \text{dw}_{pa}(V, LV, E)\),
- append \([\text{p}(C) | U]\, [\text{a}(C) | V]\, W\).

Similar predicate \(\text{dw}_{br}\) for labelled Dyck words on \{[ , ]\}.
Specification
Labelled Dyck words on \{ (, ) \}

- Labels identify matching pairs in words

\[(1(2)2(3)3)1(4)4(5(6(7)7(8)8)6)5\]
Speciﬁcation
Labelled Dyck words on \\{(, )\}\n
- Labels identify matching pairs in words
  \((1(2)2(3)3)1(4)4(5(6(7)7(8)8)8)6)5\)
- Letters encode parentheses and brackets
  
  \( '(' \rightarrow p \quad ')' \rightarrow a \quad ']' \rightarrow b \quad ']' \rightarrow r \)
Labelled Dyck words on \{\(,\)\}

- Labels identify matching pairs in words
  \((1(2)2(3)3)1(4)4(5(6)7(8)8)6)5\)
- Letters encode parentheses and brackets
  \('( → p \quad ') → a \quad '[' → b \quad ']’ → r\)
- Grammar \(D ::= \varepsilon | (D)D\)

\[
\begin{align*}
dw_{pa}([],0,\_). \\
dw_{pa}(W,L,C) & :− \text{in}(LU,0,L), \ LV \ is \ L–LU–1, \\
& \ LV \ >= \ 0, \ D \ is \ C+1, \ E \ is \ LU+D, \\
& \ dw_{pa}(U,LU,D), \ dw_{pa}(V,LV,E), \\
& \ append([p(C)|U],[a(C)|V],W).
\end{align*}
\]
Specification
Labelled Dyck words on \{ (, ) \}

- Labels identify matching pairs in words
  \((1(2)2(3)3)1(4)4(5(6)7(8)8)6)5\)
- Letters encode \textit{parentheses} and \textit{brackets}
  \('
' \rightarrow p \quad ' \rightarrow a \quad '[' \rightarrow b \quad ']' \rightarrow r\)
- Grammar \( D ::= \varepsilon | (D)D \)

\[
\begin{align*}
\text{dw}\_\text{pa}([],0,). & \\
\text{dw}\_\text{pa}(W,L,C) & := \text{in}(LU,0,L), \ LV \ \text{is} \ L-LU-1, \\
& \quad LV \geq 0, \ D \ \text{is} \ C+1, \ E \ \text{is} \ LU+D, \\
& \quad \text{dw}\_\text{pa}(U,LU,D), \ \text{dw}\_\text{pa}(V,LV,E), \\
& \quad \text{append} \left( [p(C)|U], [a(C)|V], W \right).
\end{align*}
\]

Similar predicate \texttt{dw\_br} for labelled Dyck words on \{[,[]\}\)
### Specification

Dyck word shuffles

\[
\text{shuffle} ([], [], []). \\
\text{shuffle} ([X|U], [], [X|U]). \\
\text{shuffle} ([], [X|V], [X|V]). \\
\text{shuffle} ([H|U],[K|V],[H|W]) :- \\
\quad \text{shuffle}(U,[K|V],W). \\
\text{shuffle} ([H|U],[K|V],[K|W]) :- \\
\quad \text{shuffle}([H|U],V,W). \\
\text{dws}(W,L) :- \text{in}(LU,0,L), \text{LV is } L{-}LU, \text{LV} \geq 0, \\
\text{dw_pa}(U,LU,0), \text{dw_br}(V,LV,0), \text{shuffle}(U,V,W).
\]
plw(W,N) :- dws(W,N), canonical(W).
plw(W,N) :- dws(W,N), canonical(W).

Forbidden pattern \ldots [n \ldots (m \ldots] n \ldots)m \ldots
plw(W,N) :- dws(W,N), canonical(W).

Forbidden pattern \ldots [n\ldots(m\ldots)n\ldots)m\ldots

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

noncanonical(W) :- append(C,[a(M)|\_],W),
append(B,[r(N)|\_],C), append(A,[p(M)|\_],B),
append(\_,[b(N)|\_],A).
plw(W,N) :– dws(W,N), canonical(W).

Forbidden pattern \ldots [\ldots(n\ldots)(m\ldots)n\ldots)m\ldots

\textit{canonical}(W) :– \textbackslash+ noncanonical(W).

\textit{noncanonical}(W) :– \text{append}(C, [a(M)|_], W),
append(B, [r(N)|_], C), append(A, [p(M)|_], B),
append(_, [b(N)|_], A).

Executable, thanks to Prolog backtracking mechanism
plw (W,N) :- dws(W,N), canonical(W).

Forbidden pattern \ldots [n \ldots (m \ldots )n \ldots )m \ldots 

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

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

Executable, thanks to Prolog backtracking mechanism

Not efficient, due to rejection of non-canonical shuffles
Advantages of Logic Programming

- Specification is executable
Advantages of Logic Programming

- **Specification** is executable

- Many available techniques, such as program transformations → optimization by filter promotion (e.g. [Senni and Fioravanti, TAP’12], constraint-based)
Advantages of Logic Programming

- **Specification** is executable
- Many available techniques, such as **program transformations** → optimization by filter promotion (e.g. [Senni and Fioravanti, TAP’12], constraint-based)
- Not yet achieved → Independent design of an efficient **algorithm**, and its validation wrt the former **specification**
Advantages of Logic Programming

- **Specification** is executable
- Many available techniques, such as **program transformations** → optimization by filter promotion (e.g. [Senni and Fioravanti, TAP’12], constraint-based)
- Not yet achieved → Independent design of an efficient **algorithm**, and its validation wrt the former **specification**
- A general ‘differential testing’ framework:

<table>
<thead>
<tr>
<th>generator</th>
<th>acceptor</th>
</tr>
</thead>
<tbody>
<tr>
<td>specification</td>
<td>implementation</td>
</tr>
<tr>
<td>implementation</td>
<td>specification</td>
</tr>
<tr>
<td>implementation_1</td>
<td>implementation_2</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A. Giorgetti & V. Senni
Planar Lehman word generation algorithms
Advantages of Logic Programming

- **Specification** is executable
- Many available techniques, such as program transformations $\rightarrow$ optimization by filter promotion (e.g. [Senni and Fioravanti, TAP’12], constraint-based)
- Not yet achieved $\rightarrow$ Independent design of an efficient **algorithm**, and its validation wrt the former **specification**
- A general ‘differential testing’ framework:
  
  $\begin{array}{c|c}
  \text{generator} & \text{acceptor} \\
  \text{specification} & \text{implementation} \\
  \text{specification} & \text{specification} \\
  \text{implementation} & \text{implementation} \\
  \text{implementation}_1 & \text{implementation}_2 \\
  \vdots & \vdots 
  \end{array}$
Outline

1. Introduction

2. Logical specification

3. Automata-based generation
   - Principle
   - Example
   - Specification
   - Validation
   - Performances

4. Counting

A. Giorgetti & V. Senni
Automata-based generation

For an efficient generation algorithm
Automata-based generation

For an efficient generation algorithm

- **Principle**
  - Generate words letter by letter, from left to right
  - Preserve the canonicity property during generation
Automata-based generation

For an efficient generation algorithm

- **Principle**
  - Generate words letter by letter, from left to right
  - Preserve the canonicity property during generation

- **Idea**
  - For Dyck words: count pending (unclosed) symbols (or push them on a stack)
Automata-based generation

For an efficient generation algorithm

- **Principle**
  - Generate words letter by letter, from left to right
  - Preserve the canonicity property during generation

- **Idea**
  - For Dyck words: count pending (unclosed) symbols (or push them on a stack)
  - For Dyck word shuffles: two counters? two stacks?
Automata-based generation

For an efficient generation algorithm

- **Principle**
  - Generate words letter by letter, from left to right
  - Preserve the canonicity property during generation

- **Idea**
  - For Dyck words: count pending (unclosed) symbols (or push them on a stack)
  - For Dyck word shuffles: two counters? two stacks?
  - The canonicity property can be detected with the help of a single stack of symbols
Algorithm at work

Example

Forbidden pattern \[ ( [ ] ) \]

final word \( ( [ [ ) ( [ ( ) ] ) ] ) \)

word prefix \( [ [ ) ( [ ( ) ] ) ] ) \)

stack
Forbidden pattern $[ ( ] )$

- final word $( [ [ ) ( [ ( ) ] ) ] )$
- word prefix $(
- stack $(

A. Giorgetti & V. Senni
Planar Lehman word generation algorithms
15 / 31
Forbidden pattern [ ( ] )

final word ( [ [ ) ( [ ( ) ] ] ]

word prefix ( [

stack ( [
Algorithm at work

Example

Forbidden pattern \([ ( ] )\)

final word \(( [ [ ) ( [ ( ] ] ) ] ]\)

word prefix \(( [ [\)

stack \(( [ [\)
Forbidden pattern \[ ( \[ ) \]

final word \( ( [ [ ) ( [ ( ) ] ) ] ) \)

word prefix \( ( [ [ ) \)

stack \[ [ \)
Forbidden pattern \([ ( [ ) ] )\)

final word \( ( [ [ ) ( [ ( ) ] ) ] ]\)
word prefix \( ( [ [ ) (\)
stack \( [ [ (\)

A. Giorgetti & V. Senni
Planar Lehman word generation algorithms
Algorithm at work

Example

Forbidden pattern \([ ( ] )\)

final word \(( [ [ ) ( [ ( ) ] ) ] ]\)
word prefix \(( [ [ ) ( [\)
stack \([ [ ( [\]
Algorithm at work

Example

Forbidden pattern \[ [ ( ] ) \]

final word \( ( [ [ ] ) ( [ ( ) ] ) ] ) \]

word prefix \( ( [ [ ] ) ( [ ( \]

stack \[ [ ( [ ( \]
Algorithm at work

Example

Forbidden pattern \([ ( ] )\)

final word \(( [ [ ) ( [ ( ) ] ) ] ] )\)

word prefix \(( [ [ ) ( [ (\])\)

stack \([ [ ( [\)

A. Giorgetti & V. Senni

Planar Lehman word generation algorithms

15 / 31
Forbidden pattern \([ ( ] )\)

final word \(\text{( [ ] ) ( [ ( ) ] )] ]}\)
word prefix \(\text{( [ ] ) ( [ ( ) ]}\)
stack \(\text{[ [ (}\)

Algorithm at work
Example
Forbidden pattern \([\ (\ ]\ )\ )\)

final word \((\ [\ [\ )\ (\ [\ (\ ]\ )\ )\ ]\ )\ ]\ )\]

word prefix \((\ [\ [\ )\ (\ [\ (\ ]\ )\ )\ )\ )\)

stack \([\ [\ )\ ]\)
Algorithm at work
Example

Forbidden pattern [ ( ] )

final word ( [ [ ) ( [ ( ) ] ) ] )
word prefix ( [ [ ) ( [ ( ) ] ) ]
stack [ ]
Forbidden pattern \[
[ ( ] )
\]

final word \[
( [ [ ] ( [ ( ) ] ) ] ] )
\]

word prefix \[
( [ [ ] ( [ ( ) ] ] ) ] ]
\]

stack
Forbidden pattern \([ ( ] )\)

- More than a stack: parentheses under brackets are sometimes pulled from the ‘stack’
- Counting blocks of ’[’s is sufficient \(\rightarrow\) stack of counters
Forbidden pattern \([ ( ] )\)

final word \(( [ [ ] ) ( [ ( ) ] ) ] ]\)
word prefix \(( [ [ ] ) ( [ ( ) ] ) ] ]\)
stack
counters

- More than a stack: parentheses under brackets are sometimes pulled from the ‘stack’
- Counting blocks of ’[’s is sufficient \(\rightarrow\) stack of counters
Forbidden pattern \[ ( ] ) \]

final word ( [ [ ) ( [ ( ) ] ) ] ]

word prefix
stack
counters 0

More than a stack: parentheses under brackets are sometimes pulled from the ‘stack’

Counting blocks of ’[’s is sufficient → stack of counters
Forbidden pattern \([ ( ] )\)

final word \(( [ ] ) ( [ ( ) ] ) ] ]\)

word prefix \((\)

stack \((\)

counters 0 0

More than a stack: parentheses under brackets are sometimes pulled from the ‘stack’

Counting blocks of ’[’s is sufficient → stack of counters
Forbidden pattern \([ ( ] )\)

- final word \(( [ ) ( [ ( ) ] ) ] )\)
- word prefix \(( [\)\]
- stack \(( [\)
- counters 0 1

- More than a stack: parentheses under brackets are sometimes pulled from the ‘stack’
- Counting blocks of ’[’s is sufficient \(\rightarrow\) stack of counters
Forbidden pattern $[\ (\ ]\ )$

- **final word**: $( [ [ ) ( [ ( ) ] ) ] ]$
- **word prefix**: $( [ [ )$
- **stack**: $( [ [ )$
- **counters**: $0 2$

- More than a stack: parentheses under brackets are sometimes pulled from the ‘stack’
- Counting blocks of ’[’s is sufficient $\rightarrow$ stack of counters
Forbidden pattern \([ \ ( \ ] \ )\)

- final word \((\ [\ [\ (\ [\ (\ ]\ ]\ )\ ]\ )\ ]\ )\)
- word prefix \((\ [\ [\ )\ ]\ )\)
- stack \([\ [\ )\ ]\)
- counters 2

- More than a stack: parentheses under brackets are sometimes pulled from the ‘stack’
- Counting blocks of ’[’s is sufficient \(\rightarrow\) stack of counters
Forbidden pattern \([ ( ] )\)

- **final word**: \(( [ [ ) ( [ ( ) ] ) ] ]\)
- **word prefix**: \(( [ [ ) (\)
- **stack**: \([ [ (\)
- **counters**: \(2\ 0\)

- More than a stack: parentheses under brackets are sometimes pulled from the ‘stack’
- Counting blocks of ‘[’s is sufficient \(\rightarrow\) stack of counters
Forbidden pattern \[ ( \ ] ) \]

**Algorithm at work**

**Example**

- **final word** \( ( [ [ ] ) ( [ ( ) ] ) ] ) \)
- **word prefix** \( ( [ [ ] ) ( [ \)
- **stack** \[ [ ( [ \)
- **counters** 2 1

- More than a stack: parentheses under brackets are sometimes pulled from the ‘stack’
- Counting blocks of ’[’s is sufficient \( \rightarrow \) stack of counters
Forbidden pattern \([ ([ ] ) ]\)

- final word \([ ([ ] ) ]\)
- word prefix \([ ([ ] ) ]\)
- stack \([ ([ ] ) ]\)
- counters \([2 1 0]\)

- More than a stack: parentheses under brackets are sometimes pulled from the ‘stack’
- Counting blocks of ’[’s is sufficient → stack of counters
Forbidden pattern \([ ( ] )\)

- final word: \(( [ ] ( [ ( ] ) ] ) ]\)
- word prefix: \(( [ ] ( [ ) ) ]\)
- stack: \([ [ ( [\)\])\])
- counters: 2 1

- More than a stack: parentheses under brackets are sometimes pulled from the ‘stack’
- Counting blocks of ’[’s is sufficient \(\rightarrow\) stack of counters
Forbidden pattern \([ ( ] )\)

- final word \([ [ ( ) ] ) ] ]\)
- word prefix \([ [ ( ) ] ) ]\)
- stack \([ [ ( ]\)
- counters 2 0

- More than a stack: parentheses under brackets are sometimes pulled from the ‘stack’
- Counting blocks of ’[’s is sufficient \(\rightarrow\) stack of counters
Forbidden pattern $[ ( ] )$

- final word $( [ [ ) ( [ ( ) ] ) ] ]$
- word prefix $( [ [ ) ( [ ( ) ] ) )$
- stack $[ [ )$
- counters $2$

- More than a stack: parentheses under brackets are sometimes pulled from the ‘stack’
- Counting blocks of ’[’s is sufficient $\rightarrow$ stack of counters
Forbidden pattern \([ ( ) ]\)

- final word  \(( [ [ ) ( [ ( ) ] ) ] ] )\)
- word prefix  \(( [ [ ) ( [ ( ) ] ) ] )\)
- stack  \([\)\]
- counters 1

- More than a stack: parentheses under brackets are sometimes pulled from the ‘stack’
- Counting blocks of ’[’s is sufficient \(\rightarrow\) stack of counters
Forbidden pattern \([ ( ] )\)

- final word \([ ( ] ) ( [ ( ) ] ) ] ] ]\)
- word prefix \([ ( ] ) ( [ ( ) ] ) ] ] ]\)
- counters 0

- More than a stack: parentheses under brackets are sometimes pulled from the ‘stack’
- Counting blocks of ’[’s is sufficient → stack of counters
Characteristic property of a planar Lehman word $W$

$$\text{gl0w} \ (W, \text{Size}) : - L \ \text{is} \ 2 \ast \text{Size}, \ \text{gl0w} \ (W, [0], L).$$

$$\text{gl0w} \ ([], [0], 0).$$
$$\text{gl0w} \ ([p|W], [l], L) : - L > 0, \ \text{NewL is} \ L-1,$$
$$\text{gl0w} \ (W, [0|l], \text{NewL}).$$
$$\text{gl0w} \ ([b|W], [N|l], L) : - L > 0, \ \text{NewL is} \ L-1,$$
$$\text{NewN is} \ N+1, \ \text{gl0w} \ (W, [\text{NewN}|l], \text{NewL}).$$
$$\text{gl0w} \ ([r|W], [N|l], L) : - L > 0, \ N > 0, \ \text{NewL is} \ L-1,$$
$$\text{NewN is} \ N-1, \ \text{gl0w} \ (W, [\text{NewN}|l], \text{NewL}).$$
$$\text{gl0w} \ ([a|W], [N1,N2|l], L) : - L > 0, \ \text{NewL is} \ L-1,$$
$$\text{NewN is} \ N1+N2, \ \text{gl0w} \ (W, [\text{NewN}|l], \text{NewL}).$$
Characteristic property of a planar Lehman word $W$

$gl0w(W, \text{Size}) :\neg \text{L is } 2^*\text{Size}, \quad gl0w(W,[0],\text{L}).$

$gl0w([], [0], 0).$

$gl0w([p|W], I, \text{L}) :\neg \text{L > 0, NewL is L−1},$
\quad $gl0w(W, [0|I], \text{NewL}).$

$gl0w([b|W], [N|I], \text{L}) :\neg \text{L > 0, NewL is L−1, NewN is N+1, gl0w(W, [NewN|I], \text{NewL}).}$

$gl0w([r|W], [N|I], \text{L}) :\neg \text{L > 0, N > 0, NewL is L−1, NewN is N−1, gl0w(W, [NewN|I], \text{NewL}).}$

$gl0w([a|W], [N1,N2|I], \text{L}) :\neg \text{L > 0, NewL is L−1, NewN is N1+N2, gl0w(W, [NewN|I], \text{NewL}).}$

Simple, but many failure branches (finally non-empty stacks)
Controlling the stack content with the expected word length provides efficiency

\[
\text{cl0w}(W, \text{Size}) \leftarrow L \text{ is } 2 \times \text{Size}, \text{ cl0w}(W, 0, 0, [0], L).
\]

\[
\text{cl0w}([], -, -, [0], 0).
\]

\[
\text{cl0w}([p|W], B, P, I, L) \leftarrow L > 0, L \geq B + P, \text{ NeL is } L - 1,
\]
\[
\text{NP is } P + 1, \text{ cl0w}(W, B, NP, [0|I], \text{NeL}).
\]

\[
\text{cl0w}([b|W], B, P, [C|I], L) \leftarrow L > 0, L \geq B + P, \text{ NeL is } L - 1,
\]
\[
\text{NC is } C + 1, \text{ NB is } B + 1, \text{ cl0w}(W, NB, P, [NC|I], \text{NeL}).
\]

\[
\text{cl0w}([r|W], B, P, [C|I], L) \leftarrow L > 0, L \geq B + P, \text{ C > 0, NeL is } L - 1,
\]
\[
\text{NC is } C - 1, \text{ NB is } B - 1, \text{ cl0w}(W, NB, P, [NC|I], \text{NeL}).
\]

\[
\text{cl0w}([a|W], B, P, [C1,C2|I], L) \leftarrow L > 0, L \geq B + P, \text{ NeL is } L - 1,
\]
\[
\text{NC is } C1+C2, \text{ NP is } P - 1, \text{ cl0w}(W, B, NP, [NC|I], \text{NeL}).
\]

Translated into a C program \texttt{cl0w.c}
Validation

**Input:** word size (number of symbol pairs)

- **plw**: specification
- **gl0w**: stack-based (Prolog/C) **implementation**
- **cl0w**: optimized, stack-based (Prolog/C) **implementation**
Validation

**Input:** word size (number of symbol pairs)

- **plw** : specification
- **gl0w** : stack-based (Prolog/C) implementation
- **cl0w** : optimized, stack-based (Prolog/C) implementation

**Validation**

<table>
<thead>
<tr>
<th>generator</th>
<th>acceptor</th>
</tr>
</thead>
<tbody>
<tr>
<td>plw</td>
<td>gl0w</td>
</tr>
<tr>
<td>gl0w</td>
<td>plw</td>
</tr>
<tr>
<td>plw</td>
<td>cl0w</td>
</tr>
<tr>
<td>cl0w</td>
<td>plw</td>
</tr>
</tbody>
</table>
**Validation**

**Input:** word size (number of symbol pairs)

- **plw:** specification
- **gl0w:** stack-based (Prolog/C) implementation
- **cl0w:** optimized, stack-based (Prolog/C) implementation

<table>
<thead>
<tr>
<th>generator</th>
<th>acceptor</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>plw</code></td>
<td><code>gl0w</code></td>
</tr>
<tr>
<td><code>gl0w</code></td>
<td><code>plw</code></td>
</tr>
<tr>
<td><code>plw</code></td>
<td><code>cl0w</code></td>
</tr>
<tr>
<td><code>cl0w</code></td>
<td><code>plw</code></td>
</tr>
</tbody>
</table>

Equivalence checked, incrementally, up to input size 7 (Prolog/C)
Validation

**Input:** word size (number of symbol pairs)

- **plw** : specification
- **gl0w** : stack-based (Prolog/C) implementation
- **cl0w** : optimized, stack-based (Prolog/C) implementation

**Validation**

<table>
<thead>
<tr>
<th>generator</th>
<th>acceptor</th>
</tr>
</thead>
<tbody>
<tr>
<td>plw</td>
<td>gl0w</td>
</tr>
<tr>
<td>gl0w</td>
<td>plw</td>
</tr>
<tr>
<td>plw</td>
<td>cl0w</td>
</tr>
<tr>
<td>cl0w</td>
<td>plw</td>
</tr>
</tbody>
</table>

Equivalence checked, incrementally, up to input size 7 (Prolog/C)

**Automated:** performed by LP-queries, using built-in evaluation mechanisms
### Performances

<table>
<thead>
<tr>
<th>$n$</th>
<th>$l_0(n)$</th>
<th>$t_{spec}$ (s)</th>
<th>$t_{optim}$ (s)</th>
<th>Speedup</th>
<th>$t_{rpm.c}$ (s)</th>
<th>$t_{cl0w.c}$ (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-4</td>
<td>-</td>
<td>0.00</td>
<td>0.00</td>
<td>NS</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>2,916</td>
<td>0.03</td>
<td>0.01</td>
<td>3.0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>24,057</td>
<td>0.41</td>
<td>0.08</td>
<td>5.1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>208,494</td>
<td>5.13</td>
<td>0.65</td>
<td>7.9</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>1,876,446</td>
<td>68.95</td>
<td>5.99</td>
<td>11.5</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>17,399,772</td>
<td>923.94</td>
<td>55.20</td>
<td>16.7</td>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>165,297,834</td>
<td>14,255.00</td>
<td>575.00</td>
<td>24.8</td>
<td>95</td>
<td>12</td>
</tr>
<tr>
<td>11</td>
<td>1,602,117,468</td>
<td>too long</td>
<td>5,227.00</td>
<td>-</td>
<td>1005</td>
<td>125</td>
</tr>
</tbody>
</table>

**Specification:** Obvious correctness, **no** efficiency

**Optimized (automata-based):** Efficient, **no** correctness proof
Outline

1. Introduction
2. Logical specification
3. Automata-based generation
4. Counting
   - Matching pair removal
   - Validation
   - Formal proof
5. Conclusion
Counting
From the automata-based algorithm

Let $n_{n_1,\ldots,n_{r+1},l}$ be the number of PLWs appending some word of length $l$ to some word whose subword of pending symbols is $[n_1 \ldots [n_{r+1}$.

The generating function

$$N(x, t) = \sum_{l,r \geq 0} \sum_{n_1,\ldots,n_{r+1} \geq 0} n_{n_1,\ldots,n_{r+1},l} u_1^{n_1} \ldots u_{r+1}^{n_{r+1}} x^l t^r$$

is the unique formal power series solution of the equation

$$N(x, t) = 1 + x(t^{-1} (N(x, t) - N(x, 0))|_{u_1 := 0, \forall i \geq 2}.u_i := u_{i-1})$$
$$+ u_1^{-1} (N(x, t) - N(x, t)|_{u_1 := 0}) + u_1 N(x, t)$$
$$+ t (u_2 - u_1)^{-1} (u_2 N(x, t)|_{\forall i \geq 1}.u_i := u_{i+1}$$
$$- u_1 N(x, t)|_{\forall i \geq 2}.u_i := u_{i+1})).$$
Another decomposition of planar Lehman words
By matching pair removal
Another decomposition of planar Lehman words
By matching pair removal

Let $\mathcal{P}_2$ be the set of words on $\{(,), [, ]\}$ composed of $\varepsilon$, the words $[u]v$ whenever $u$ and $v$ are in $\mathcal{P}_2$, and the words $(u)v$ whenever $u \circ v$ is in $\mathcal{P}_2$ and the restriction of $u$ to $\{(, )\}$ is a Dyck word.
Another decomposition of planar Lehman words
By matching pair removal

Let $\mathcal{P}_2$ be the set of words on $\{(,), [, ]\}$ composed of $\varepsilon$, the words $[u]v$ whenever $u$ and $v$ are in $\mathcal{P}_2$, and the words $(u)v$ whenever $u \circ v$ is in $\mathcal{P}_2$ and the restriction of $u$ to $\{(,)\}$ is a Dyck word.

**Theorem 1 [Cori75, Property II.7]**

$\mathcal{P}_2$ and the set $\mathcal{P}_1$ of planar Lehman words are in length-preserving one-to-one correspondence.
Another decomposition of planar Lehman words
By matching pair removal

Let $\mathcal{P}_2$ be the set of words on \{(,),[,]\} composed of $\varepsilon$, the words $[u]v$ whenever $u$ and $v$ are in $\mathcal{P}_2$, and the words $(u)v$ whenever $u \circ v$ is in $\mathcal{P}_2$ and the restriction of $u$ to \{(,\}\} is a Dyck word.

**Theorem 1 [Cori75, Property II.7]**

$\mathcal{P}_2$ and the set $\mathcal{P}_1$ of planar Lehman words are in length-preserving one-to-one correspondence.

**Definition**

The cut number $c(w)$ of a word $w \in \{(,),[,]\}^*$ is the number of its prefixes whose restriction to parentheses is a Dyck word. By convention, $c(\varepsilon) = 1$. 
Let \( l_0(n) \) (resp. \( l_0(n, k) \)) be the number of planar Lehman words of size \( n \) (resp. with cut number \( k \)).

**Proposition**

\[
L_0(x) = \sum_{n \geq 0} l_0(n)x^n \quad \text{and} \quad M(x, y) = \sum_{n \geq 1} \sum_{k=2}^{2n+1} l_0(n, k)x^{n-1}y^{k-1}
\]
satisfy

\[
\begin{align*}
L_0(x) &= 1 + xM(x, 1) \\
M(x, y) &= y + y^2 + xy(1 - y)^{-1}(M(x, 1) - y^2M(x, y)) \\
&\quad + xy^2M(x, y) + x^2y^2M(x, y)^2
\end{align*}
\]

\[
\rightarrow l_0(n) = \frac{2(2n)!3^n}{n!(n+2)!} \quad [\text{Tutte63}]
\]
## Validation

<table>
<thead>
<tr>
<th>n</th>
<th>$l_0(n)$</th>
<th>Prolog</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>spec.</td>
<td>optim.</td>
<td>pair removal</td>
<td></td>
</tr>
<tr>
<td>0-4</td>
<td>-</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>2,916</td>
<td>0.03</td>
<td>0.01</td>
<td>0.00</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>24,057</td>
<td>0.41</td>
<td>0.08</td>
<td>0.06</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>208,494</td>
<td>5.13</td>
<td>0.65</td>
<td>0.65</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1,876,446</td>
<td>68.95</td>
<td>5.99</td>
<td>7.23</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>17,399,772</td>
<td>923.94</td>
<td>55.20</td>
<td>81.47</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>165,297,834</td>
<td>14,255.00</td>
<td>575.00</td>
<td>910.22</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>1,602,117,468</td>
<td>too long</td>
<td>5,227.00</td>
<td>10,129.66</td>
<td></td>
</tr>
</tbody>
</table>

Proper decomposition $\Rightarrow$ declarativeness and better efficiency

How to discover good decompositions?
Theorem 1 (reminder)

$P_2$ and the set $P_1$ of planar Lehman words are in length-preserving one-to-one correspondence.
Theorem 1 (reminder)

$\mathcal{P}_2$ and the set $\mathcal{P}_1$ of planar Lehman words are in length-preserving one-to-one correspondence.

- A paper-and-pen proof fits in two pages
  [Cori75, pages 134-135]
Theorem 1 (reminder)

\( P_2 \) and the set \( P_1 \) of planar Lehman words are in length-preserving one-to-one correspondence.

- A paper-and-pen proof fits in two pages [Cori75, pages 134-135]
- How hard is it to prove this theorem formally?
Theorem 1 (reminder)

$P_2$ and the set $P_1$ of planar Lehman words are in length-preserving one-to-one correspondence.

- A paper-and-pen proof fits in two pages [Cori75, pages 134-135]
- How hard is it to prove this theorem formally?
- First experiment with Coq (proof assistant, calculus of inductive constructions, based on the theory of dependent types)
Some Coq code

Definition of $\mathcal{P}_2$

$\mathcal{P}_2$ is the set of words on $\{(,), [],\}$ composed of $\varepsilon$, the words $[u]v$ whenever $u$ and $v$ are in $\mathcal{P}_2$ and the words $(u)v$ whenever $u \circ v$ is in $\mathcal{P}_2$ and the restriction of $u$ to $\{(,)\}$ is a Dyck word.
Some Coq code

Definition of $\mathcal{P}_2$

$\mathcal{P}_2$ is the set of words on $\{(,),[[],]\}$ composed of $\varepsilon$, the words $[u]v$ whenever $u$ and $v$ are in $\mathcal{P}_2$ and the words $(u)v$ whenever $u \circ v$ is in $\mathcal{P}_2$ and the restriction of $u$ to $\{(,())\}$ is a Dyck word.

```coq
Inductive plw2 : nat -> nat -> nat -> nat -> word -> Prop :=
| plw2mty : forall n m : nat, plw2 (S n) n (S m) m nil
| plw2bracket :
  forall (fp lp fb lb lpu lbu : nat) (u v : word),
  plw2 fp lpu (S fb) lbu u ->
  plw2 (S lpu) lp (S lbu) lb v ->
  plw2 fp lp fb lb (B fb :: u ++ R fb :: v)
| plw2paren :
  forall (fp lp fb lb lpu lbu : nat) (u v : word),
  dwpa (S fp) lpu (rmBR u) ->
  plw2 (S fp) lp fb lb (u ++ v) ->
  plw2 fp lp fb lb (P fp :: u ++ A fp :: v).
```
Some Coq code

Definition of $P_2$

$P_2$ is the set of words on $\{ (,) , [ , ] \}$ composed of $\varepsilon$, the words $[u]v$ whenever $u$ and $v$ are in $P_2$ and the words $(u)v$ whenever $u \circ v$ is in $P_2$ and the restriction of $u$ to $\{ (,) \}$ is a Dyck word.

\[
\text{Inductive } \text{plw2 : nat} \rightarrow \text{nat} \rightarrow \text{nat} \rightarrow \text{nat} \rightarrow \text{word} \rightarrow \text{Prop} := \\
| \text{plw2mty :} \\
| \quad \text{forall n m : nat, plw2} (\text{S n}) n (\text{S m}) m \text{ nil} \\
| \text{plw2bracket :} \\
| \quad \text{forall (fp lp fb lb lpu lbu : nat) (u v : word),} \\
| \quad \text{plw2} \text{ fp lp lpu} (\text{S fb}) \text{ lbu u} \rightarrow \\
| \quad \text{plw2} \text{ (S lpu)} lp (\text{S lbu}) lb v \rightarrow \\
| \quad \text{plw2} \text{ fp lp fb lb (B fb :: u ++ R fb :: v)} \\
| \text{plw2paren :} \\
| \quad \text{forall (fp lp fb lb lpu lbu : nat) (u v : word),} \\
| \quad \text{dwpa} (\text{S fp}) lpu (\text{rmBR u}) \rightarrow \\
| \quad \text{plw2} \text{ (S fp)} lp fb lb (u ++ v) \rightarrow \\
| \quad \text{plw2} \text{ fp lp fb lb (P fp :: u ++ A fp :: v)}. \\
\]

With labels

A. Giorgetti & V. Senni
Planar Lehman word generation algorithms

26 / 31
Definition of $\mathcal{P}_1$

**Inductive** plw1 : nat -> nat -> nat -> nat -> word -> Prop :=

| plw1ax :  
|   forall fp lp fb lb : nat, forall w : word, 
|   dws fp lp fb lb w -> canonical w -> plw1 fp lp fb lb w.

**Main lemma (one-half of Theorem 1)**

**Lemma** imp21 (fp lp fb lb : nat) (w : word) : 
plw2 fp lp fb lb w -> plw1 fp lp fb lb w.
Proving experience

- Metrics

<table>
<thead>
<tr>
<th>Subject</th>
<th>Nb. of lines for defs.</th>
<th>Nb. of lemmas</th>
<th>Nb. of lines of proof</th>
</tr>
</thead>
<tbody>
<tr>
<td>words (lists)</td>
<td>7</td>
<td>6</td>
<td>56</td>
</tr>
<tr>
<td>Dyck words</td>
<td>6 (× 2)</td>
<td>14</td>
<td>216</td>
</tr>
<tr>
<td>Shuffles</td>
<td>12 (+16+2)</td>
<td>32</td>
<td>213</td>
</tr>
<tr>
<td>$\mathcal{P}_1$</td>
<td>8</td>
<td>3</td>
<td>384</td>
</tr>
<tr>
<td>$\mathcal{P}_2$</td>
<td>13</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>$\mathcal{P}_2 \subseteq \mathcal{P}_1$</td>
<td>-</td>
<td>1</td>
<td>29</td>
</tr>
<tr>
<td>Totals</td>
<td>57</td>
<td>56</td>
<td>898</td>
</tr>
</tbody>
</table>

- Two families of Dyck words $\rightarrow$ Many lemmas repeated twice
- Suggests a more practical characterization of Dyck word shuffles: The restriction to each alphabet is a Dyck word
Outline

1. Introduction
2. Logical specification
3. Automata-based generation
4. Counting
5. Conclusion
Two characterizations of planar Lehman words
- Validated by bounded-exhaustive testing
  (up to size 7, > 200,000 cases)
Two characterizations of planar Lehman words

- Validated by bounded-exhaustive testing
  (up to size 7, > 200,000 cases)
- “A proper decomposition implies the definition” is formally proved (the reverse implication is ongoing work)
Two characterizations of planar Lehman words
  Validated by bounded-exhaustive testing
    (up to size 7, > 200,000 cases)
  “A proper decomposition implies the definition” is formally proved (the reverse implication is ongoing work)
Efficient planar Lehman word generation algorithms
Conclusion

Summary

- Two characterizations of planar Lehman words
  - Validated by bounded-exhaustive testing (up to size 7, > 200,000 cases)
  - “A proper decomposition implies the definition” is formally proved (the reverse implication is ongoing work)
- Efficient planar Lehman word generation algorithms
- Differential testing library for logical specifications
Conclusion

Summary

- Two characterizations of planar Lehman words
  - Validated by bounded-exhaustive testing
    (up to size 7, > 200,000 cases)
  - “A proper decomposition implies the definition” is formally proved (the reverse implication is ongoing work)

- Efficient planar Lehman word generation algorithms

- Differential testing library for logical specifications

http://www.disp.uniroma2.it/users/senni/validation.html
Two characterizations of planar Lehman words
- Validated by bounded-exhaustive testing (up to size 7, > 200,000 cases)
- “A proper decomposition implies the definition” is formally proved (the reverse implication is ongoing work)

Efficient planar Lehman word generation algorithms

Differential testing library for logical specifications

http://www.disp.uniroma2.it/users/senni/validation.html
A formal methodology for combinatorial object/data structures generation and counting
A formal methodology for combinatorial object/data structures generation and counting

“Think and specify, test and prove”
Conclusion

Perspectives

- A formal methodology for combinatorial object/data structures generation and counting

  “Think and specify, test and prove”

- Optimization (filter promotion) by general-purpose formal transformations

A. Giorgetti & V. Senni
A formal methodology for combinatorial object/data structures generation and counting

“Think and specify, test and prove”

Optimization (filter promotion) by general-purpose formal transformations

Application to the formal specification of rooted maps and hypermaps of any positive genus (generation is validated, counting is ongoing work)