Specification and Validation of Algorithms Generating Planar Lehman Words

Alain Giorgetti Valerio Senni

University of Franche-Comté
FEMTO-ST
Inria, CASSIS project

University of Rome “Tor Vergata”
Dept. of Computer Science,
Systems and Production

GASCom 2012

Introduction
Motivations

D. Zeilberger: "Teach your computer how to do research!"

Introduction
Motivations

D. Zeilberger: "Teach your computer how to do research!"

- Computer-assisted design and validation of efficient enumeration algorithms

Introduction
Motivations

D. Zeilberger: "Teach your computer how to do research!"

- Computer-assisted design and validation of efficient enumeration algorithms
  - From a high-level (formal) specification

Introduction
Motivations

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

Introduction
Motivations

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

Introduction
Motivations

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)

Introduction
Motivations

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

Combinatorial case study

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

Dyck words
Definition

- A Dyck word of length $2n$ (size n) contains n pairs of parentheses (possibly nested) which correctly match

Dyck words
Definition

- 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: $D ::= \varepsilon \mid (\, D \,)\, D$

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 $\ldots[\ldots(\ldots]\ldots)\ldots$
- 9 planar Lehman words with 4 letters:
  $(())\quad([])\quad([)]\quad()()\quad()[]\quad[()]\quad[[]]\quad[]()\quad[][]$
- 1 noncanonical Dyck word shuffle with 4 letters: $[(])$

Outline

Logic programming

- Programs are sets of rules (Horn clauses) of the form
  $\mathtt{H} \ :\!- \ \mathtt{A}_1 \wedge \ldots \wedge \mathtt{A}_n$
  (meaning, $\mathtt{H}$ holds if $\mathtt{A}_i$ holds for $i = 1, \ldots, n$)

Logic programming

- Programs are sets of rules (Horn clauses) of the form

  $\texttt{H} \texttt{ :- } \texttt{A}_1 \wedge \ldots \wedge \texttt{A}_n$

  (meaning, $\texttt{H}$ holds if $\texttt{A}_i$ holds for $i = 1, \ldots, n$)
- Example

```
ordered([]).
ordered([X]).
ordered([X1,X2|L]) :- X1 ≤ X2 ∧ ordered([X2|L]).
```

Logic programming

- Programs are sets of rules (Horn clauses) of the form
  $\mathtt{H} \ \mathtt{:-} \ \mathtt{A}_1 \wedge \ldots \wedge \mathtt{A}_n$
  (meaning, $\mathtt{H}$ holds if $\mathtt{A}_i$ holds for $i = 1, \ldots, n$)
- Example

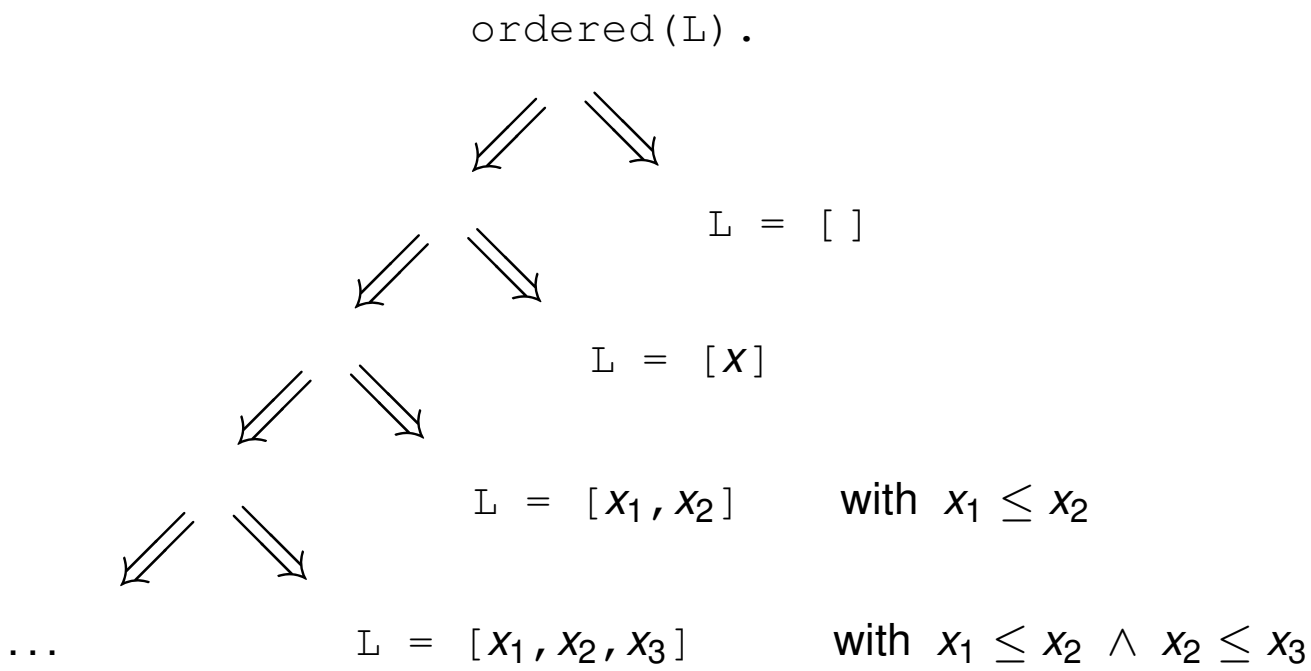
```
ordered([]).
ordered([X]).
ordered([X1,X2|L]) :- X1 ≤ X2 ∧ ordered([X2|L]).
```

- Query evaluation
  1. Pick leftmost atom in current query: $\mathtt{Q} = \mathtt{A} \wedge \mathtt{R}$
  2. Find unifying head: $\mathtt{A}\,\sigma = \mathtt{H}\,\sigma$
  3. Rewrite to get a new query: $(\mathtt{A}_1 \wedge \ldots \wedge \mathtt{A}_n \wedge \mathtt{R})\,\sigma$

Logic Programming
Generation

```
ordered([]).
ordered([X]).
ordered([X1,X2|L]) :- X1 ≤ X2 ∧ ordered([X2|L]).
```

as a generator:



Specification

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 $\{(,)\}$

- Labels identify matching pairs in words

$$(_1(_2)_2(_3)_3)_1(_4)_4(_5(_6(_7)_7(_8)_8)_6)_5$$

- Letters encode `pa`rentheses and `br`ackets

$$'(' \to \texttt{p} \qquad ')' \to \texttt{a} \qquad '[' \to \texttt{b} \qquad ']' \to \texttt{r}$$

Specification

Labelled Dyck words on $\{(,)\}$

- Labels identify matching pairs in words

$$(_1(_2)_2(_3)_3)_1(_4)_4(_5(_6(_7)_7(_8)_8)_6)_5$$

- Letters encode `pa`rentheses and `br`ackets

$$'(' \to \texttt{p} \qquad ')' \to \texttt{a} \qquad '[' \to \texttt{b} \qquad ']' \to \texttt{r}$$

- Grammar $D ::= \varepsilon \mid (\ D\ )\ D$

```
dw_pa([],0,_).
dw_pa(W,L,C) :- 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).
```

Specification

Labelled Dyck words on $\{(,)\}$

- Labels identify matching pairs in words

$$(_1(_2)_2(_3)_3)_1(_4)_4(_5(_6(_7)_7(_8)_8)_6)_5$$

- Letters encode parentheses and brackets

$$'(' \to \texttt{p} \qquad ')' \to \texttt{a} \qquad '[' \to \texttt{b} \qquad ']' \to \texttt{r}$$

- Grammar $D ::= \varepsilon \mid (\, D \,)\, D$

```
dw_pa([],0,_).
dw_pa( W,L,C) :- 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).
```

Similar predicate dw_br for labelled Dyck words on $\{[,]\}$

Specification
Dyck word shuffles

```
shuffle(    [],    [],    []).
shuffle([X|U],    [],[X|U]).
shuffle(    [],[X|V],[X|V]).
shuffle([H|U],[K|V],[H|W]) :-
  shuffle(U,[K|V],W).
shuffle([H|U],[K|V],[K|W]) :-
  shuffle([H|U],V,W).
```

```
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).
```

Specification
Planar Lehman words

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

Specification
Planar Lehman words

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

- Forbidden pattern $\ldots[_n\ldots(_m\ldots]_n\ldots)_m\ldots$

Specification
Planar Lehman words

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

- Forbidden pattern $\ldots[_n\ldots(_m\ldots]_n\ldots)_m\ldots$

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

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

Specification
Planar Lehman words

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

- Forbidden pattern $\ldots[_n\ldots(_m\ldots]_n\ldots)_m\ldots$

```
canonical(W) :- \+ 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

Specification
Planar Lehman words

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

- Forbidden pattern $\ldots[_n\ldots(_m\ldots]_n\ldots)_m\ldots$

```
canonical(W) :- \+ 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:

generator	*acceptor*
specification	implementation
implementation	specification
implementation_1	implementation_2
⋮	⋮

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:

generator	*acceptor*
specification	implementation
implementation	specification
implementation_1	implementation_2
⋮	⋮

Outline

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

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	(	[	[										

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	[	[	(	[								

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		[	[	(								

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												

Algorithm at work
Example

Forbidden pattern [( ])

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

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

Algorithm at work
Example

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

Algorithm at work
Example

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 $\rightarrow$ stack of counters

Algorithm at work
Example

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 $\rightarrow$ stack of counters

Algorithm at work
Example

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

Algorithm at work
Example

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

Algorithm at work
Example

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

Algorithm at work
Example

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

Algorithm at work
Example

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

Algorithm at work
Example

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 $\rightarrow$ stack of counters

Algorithm at work
Example

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

Algorithm at work
Example

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

Algorithm at work
Example

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

Algorithm at work
Example

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

Algorithm at work
Example

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 $\rightarrow$ stack of counters

Automata-based generation

Logical specification

Characteristic property of a planar Lehman word W

```
glow(W,Size) :- L is 2*Size, glow(W,[0],L).

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

Automata-based generation

Logical specification

Characteristic property of a planar Lehman word W

```
glow(W,Size) :- L is 2*Size, glow(W,[0],L).

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

Simple, but many failure branches (finally non-empty stacks)

Automata-based generation

Optimization

Controlling the stack content with the expected word length provides efficiency

```
clow(W,Size) :- L is 2*Size, clow(W,0,0,[0],L).

clow([],_,_,[0],0).
clow([p|W],B,P,I,L) :- L>0, L>=B+P, NeL is L-1,
  NP is P+1, clow(W, B,NP,[0|I],NeL).
clow([b|W],B,P,[C|I],L) :- L>0, L>=B+P, NeL is L-1,
  NC is C+1, NB is B+1, clow(W,NB, P,[NC|I],NeL).
clow([r|W],B,P,[C|I],L) :- L>0, L>=B+P, C>0, NeL is L-1,
  NC is C-1, NB is B-1, clow(W,NB, P,[NC|I],NeL).
clow([a|W],B,P,[C1,C2|I],L) :- L>0, L>=B+P, NeL is L-1,
  NC is C1+C2, NP is P-1, clow(W, B,NP,[NC|I],NeL).
```

Translated into a C program **`clow.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

generator	*acceptor*
`plw`	`gl0w`
`gl0w`	`plw`
`plw`	`cl0w`
`cl0w`	`plw`

Validation

Input: word size (number of symbol pairs)

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

generator	*acceptor*
`plw`	`gl0w`
`gl0w`	`plw`
`plw`	`cl0w`
`cl0w`	`plw`

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

generator	*acceptor*
`plw`	`gl0w`
`gl0w`	`plw`
`plw`	`cl0w`
`cl0w`	`plw`

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

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

Performances

		Prolog			C	
		spec.	optim.	speedup	`rpm.c`	`c10w.c`
n	$l_0(n)$	time (s)	time (s)	factor	time (s)	time (s)
0-4	-	0.00	0.00	NS	0	0
5	2,916	0.03	0.01	3.0	0	0
6	24,057	0.41	0.08	5.1	0	0
7	208,494	5.13	0.65	7.9	0	0
8	1,876,446	68.95	5.99	11.5	0	0
9	17,399,772	923.94	55.20	16.7	9	1
10	165,297,834	14,255.00	575.00	24.8	95	12
11	1,602,117,468	too long	5,227.00	-	1005	125

Specification: Obvious correctness, **no** efficiency
Optimized (automata-based): Efficient, **no** correctness proof

Outline

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

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

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 ε, 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 ε, 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 ε, 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, $c(\varepsilon) = 1$.

Counting
New functional equations

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

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

$$\begin{aligned} L_0(x) &= 1 + xM(x,1) \\ M(x,y) &= y + y^2 + xy(1-y)^{-1}(M(x,1) - y^2M(x,y)) \\ &\quad + xy^2M(x,y) + x^2y^2M(x,y)^2 \end{aligned}$$

$\rightarrow l_0(n) = \frac{2(2n)!3^n}{n!(n+2)!}$ [Tutte63]

Validation

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

Proper decomposition $\Rightarrow$ declarativeness **and** better efficiency

How to discover good decompositions?

Formal proof
Principle

Theorem 1 (reminder)

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

Formal proof
Principle

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]

Formal proof
Principle

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]
- How hard is it to prove this theorem formally?

Formal proof
Principle

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]
- 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 ε, 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 ε, 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.

```
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 $\mathcal{P}_2$

$\mathcal{P}_2$ is the set of words on $\{(,),[,]\}$ composed of ε, 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.

```
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).
```

With labels

Some Coq code

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

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

- 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

Conclusion
Summary

- Two characterizations of planar Lehman words
  - Validated by bounded-exhaustive testing
    (up to size 7, $>$ 200,000 cases)

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)

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

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

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

Conclusion
Perspectives

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

Conclusion
Perspectives

- 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

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