Sequential generation of structured arrays and its deductive verification

Richard Genestier[1] Alain Giorgetti[1,2] Guillaume Petiot[1,3]

[1]FEMTO-ST institute (UMR CNRS 6174 - UBFC/UFC/ENSMM/UTBM)
University of Franche-Comté
[2]INRIA Nancy Grand-Est, CASSIS team
[3]CEA, LIST, Software Reliability Laboratory

TAP 2015

Introduction

- Motivations
  - Can we trust our verification or testing tools?
  - Build verification environments that are themselves certified
  - Focus on exhaustive generation of structured data (for bounded-exhaustive testing)
- Present work
  - Algorithms from enumerative combinatorics
  - Combinatorial structures stored in a C array satisfying given structural constraints
- Notion of sequential generator
  - Two C functions, generating all the arrays with a given size, one after another, in a total order
  - int first_x(int a[], int n,...) generates the first array a of size n in the family x
  - int next_x(int a[], int n, ...) generates in the array a of size n the next element of the family x, immediately following the one stored in the array a when the function is called
- Expected properties
  - Soundness: each generated array satisfies its structural constraints
  - Progress: each generated array is greater than the previous one
  - Exhaustivity: all the arrays are generated

Tools: Frama-C + plugins

- C code analysis framework developed by CEA LIST and INRIA Saclay
- Specification language ACSL annotating C programs
- WP plugin for Weakest Precondition calculus
- Generation of verification conditions (first-order logic) with Why3
- Calls SMT solvers (Alt-Ergo, CVC3, CVC4)
- Stady plugin (developed by G. Petiot) for dynamic analysis

Outline

Outline

RGF

Restricted growth functions (RGF)

A *restricted growth function* (RGF, for short) of size n is a function a from $\{0, \ldots, n-1\}$ to $\{0, \ldots, n-1\}$ such that $a(0) = 0$ and $a(i) \leq a(i-1) + 1$ for $1 \leq i \leq n-1$.

- Represented by a C array of values:

0	1	...	$n-1$
$a(0)$	$a(1)$	...	$a(n-1)$

- Example: [0 1 1 0 1 2 3] is a RGF of size 7, but [1 1 2 0 1 2 1] and [0 1 2 1 3 3 2] are not.

```
/*@ predicate is_non_neg(int *a, integer n) =
  @  \forall integer i; 0 <= i < n ==> a[i] >= 0;
  @ predicate is_le_pred(int *a, integer n) =
  @  \forall integer i; 1 <= i < n ==> a[i] <= a[i-1]+1;
  @ predicate is_rgf(int *a, integer n) =
  @  is_non_neg(a,n) && a[0] == 0 && is_le_pred(a,n); */
```

Efficient generation of RGFs

Generation algorithm [Arn10, page 235]

- In increasing order, the first RGF of size n is '0'^{n} = | 0 | 0 | ... | 0 |
- The successor of the RGF a is computed by incrementing the rightmost value $a(j)$ such that $a(j) \leq a(j-1)$ and then setting $a(i) = 0$ for all $i > j$

Example:

0	1	2	2	0	1	2	3
0	1	2	2	0	1	2	3
0	1	2	2	1			
0	1	2	2	1	0	0	0

```
for (i = n-1; i >= 1; i--) if (a[i] <= a[i-1]) break;
a[i]++;
for (k = i+1; k < n; k++) a[k] = 0;
```

We implement these three steps in a function named `next_rgf`

ACSL specification of next_rgf

```
/*@ requires n > 0 && \valid(a+(0..n-1)) && is_rgf(a,n);
  @ assigns a[1..n-1];
  @ ensures is_rgf(a,n); */
int next_rgf(int a[], int n) {
  int rev,k;
  /*@ loop invariant 0 <= rev <= n-1;
    @ loop assigns rev;
    @ loop variant rev; */
  for (rev = n-1; rev >= 1; rev--) if (a[rev] <= a[rev-1]) break;
  if (rev == 0) return 0; // Last RGF.
  a[rev]++;
  /*@ loop invariant rev+1 <= k <= n;
    @ loop invariant is_non_neg(a,k) && is_le_pred(a,k);
    @ loop assigns k, a[rev+1..n-1];
    @ loop variant n-k; */
  for (k = rev+1; k < n; k++) a[k] = 0;
  return 1;
}
```

Progress property

Lexicographic order

The *lexicographic order* on arrays b and c of size n is the binary relation $\prec$ such that $b \prec c$ if and only if there is an index i ($0 \leq i < n$) such that

- $b[j] = c[j]$ for $0 \leq j \leq i - 1$
- $b[i] < c[i]$

- Example:

0	1	2	2	3	4	$\prec$	0	1	2	3	0	0

- In ACSL, `\at(e,L)` is the value of the expression `e` at label `L`
- Label `Pre` (resp. `Post`) before (resp. after) the execution of `next_rgf`

```
/*@ ensures \result == 1 ==> lt_lex{Pre,Post}(a,n); */
int next_rgf(int a[], int n) { ...
```

```
/*@ predicate lt_lex{L1,L2}(int *a, integer n) =
  @  \exists int i; 0 <= i < n && is_eq{L1,L2}(a,i) &&
  @  \at(a[i],L1) < \at(a[i],L2); */
```

Outline

Generation patterns

For a family `x`, a generation pattern for a sequential generator in lexicographic order is a C code and ACSL annotations for functions `first_x` and `next_x`

```
/*@ requires n > 0 && \valid(a+(0..n-1));
  @ assigns a[0..n-1];
  @ ensures is_x(a,n); */
int first_x(int a[], int n);

/*@ requires n > 0 && \valid(a+(0..n-1)) && is_x(a,n);
  @ assigns a[0..n-1];
  @ ensures is_x(a,n);
  @ ensures \result == 1 ==> lt_lex{Pre,Post}(a,n); */
int next_x(int a[], int n);
```

Pattern of function next_x with suffix revision

```
int next_x(int a[], int n) {
  int rev;
  // 1.  Search of the revision index rev, from right to left
  /*@ loop invariant -1 <= rev <= n-1;
    @ loop invariant
       \forall integer j; rev < j < n ==> ! is_rev(a,n,j);
    @ loop assigns rev;
    @ loop variant rev; */
  for (rev = n-1; rev >= 0; rev--) if (b_rev(a,n,rev)) break;
  // 2.  If no revision index, last array reached
  if (rev == -1) return 0;
  // 3.  Suffix revision from left to right, from rev
  suffix(a,n,rev);
  return 1;
}
```

with

```
/*@ ensures \result == 1 <==> is_rev(a,n,rev); */
int b_rev(int a[], int n, int rev);
```

Generation by filtering

- Structured arrays defined from general arrays by a characteristic constraint
- Generation by filtering consists of selecting among some arrays those that satisfy a given constraint

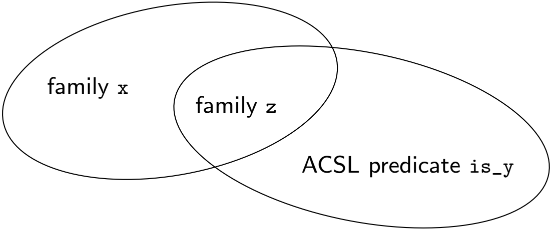
Example: RGF family

- Subfamily of the family of endofunctions of $\{0, \ldots, n-1\}$
- From `first_endofct(a,n)` and `next_endofct(a,n)`
- Filtering those endofunctions of $\{0, \ldots, n-1\}$ that are RGFs
- C Boolean function `b_rgf`: returns 1 if the endofunction is a RGF, and 0 otherwise

ACSL specification of next_rgf by filtering

```
/*@ requires n > 0 && \valid(a+(0..n-1)) && is_rgf(a,n);
  @ assigns  a[0..n-1];
  @ ensures  \result == 0 || \result == 1;
  @ ensures  \result == 1 ==> is_rgf(a,n);
  @ ensures \result == 1 ==> lt_lex{Pre,Post}(a,n); */
int next_rgf(int a[], int n) {
  int tmp = 0;
  /*@ loop assigns  a[0..n-1], tmp;
    @ loop invariant  is_endofct(a,n);  */
  do {
    tmp = next_endofct(a,n);
  } while (tmp != 0 && b_rgf(a,n) == 0);
  if (tmp == 0) { return 0; }
  return 1;
}
```

General pattern for generation by filtering



- Generation of arrays of family z by filtering arrays of family x and selecting those satisfying the characteristic constraint is_y
- If first_x(a,n), next_x(a,n) and b_y(a,n) are verified, first_z(a,n) and next_z(a,n) are automatically verified
- ```
  /*@ ensures \result == 1 <==> is_y(a,n);  */
  int b_y(int a[], int n);
  ```
- General translation rules of the first-order predicate is_y into the C Boolean function b_y
  - Automated verification of b_y
  - Patterns for predicates with nested quantifiers: $\forall\exists$, $\exists\forall$ and $\forall\forall$
  ```

Outline

Patterns

Implementation, specification and automated verification of patterns of sequential generation algorithms by suffix revision, by filtering and Boolean functions

Computation time limited to 2 minutes

Example	C code	ACSL	goals	Alt-Ergo (s)
suffix	9	12	26	2.873
filtering	14	33	51	1.230
allex	11	28	40	0.557
exall	12	27	40	0.545
all2	40	28	40	0.577

Generation by filtering

- Generation of subfamilies of the family fct generating functions from $\{0, \ldots, n-1\}$ to $\{0, \ldots, k-1\}$
- Using filtering and Boolean function patterns
  - Family rgf of restricted growth functions on $\{0, ..., n-1\}$
  - Family comb of combinations of p elements selected from n
  - Family sorted of sorted arrays of length n
  - Family inj of injections from $\{0, ..., n-1\}$ to $\{0, ..., k-1\}$ $(k \geq n)$
  - Family surj of surjections from $\{0, ..., n-1\}$ to $\{0, ..., k-1\}$ $(k \leq n)$
  - Family perm of permutations of n elements
  - Family invol of involutions of n elements
  - Family derang of derangements of n elements

Example	C code	ACSL	goals	Alt-Ergo (s)	CVC3 (s)
rgf $\subset$ endofct	25	27	69	1.340	3.524
comb $\subset$ fct	21	28	67	Timeout	3.863
sorted $\subset$ fct	19	27	67	1.212	3.604
inj $\subset$ fct	29	42	91	1.842	4.512
surj $\subset$ fct	29	40	103	1.723	4.797
perm $\subset$ fct	30	42	91	1.493	4.413
perm $=$ endofct $\wedge$ inj	17	21	60	1.122	3.499
perm $=$ endofct $\wedge$ surj	28	40	102	1.595	4.501
invol $\subset$ perm	20	27	66	1.458	3.976
derang $\subset$ perm	20	27	66	1.440	3.942

Generation by suffix revision

- Generators of the families
  - fct: functions from $\{0, \ldots, n-1\}$ to $\{0, \ldots, k-1\}$
  - subset: subsets of a set of n elements
- More efficient generators of the families rgf, sorted, comb and perm

Example	C code	ACSL	goals	Alt-Ergo + CVC3+CVC4 (s)	+ final assertion (s)
fct	13	26	43	6.774	6.858
subset	13	22	40	6.774	6.428
rgf	13	28	41	7.741	8.359
sorted	13	30	44	27.607	8.448
comb	18	33	46	Timeout	29.379
perm	23	29	50	12.366	10.778

- Final assertion `/*@ assert is_eq{Pre,Here}(a,rev) && \at(a[rev],Pre) < a[rev]; */` to speed up the proof for the progress property

Validations

Soundness and progress properties proved

How to check exhaustivity?

- Validation by increasing size, up to some size, by counting the number of generated arrays
- Compared to the expected number obtained thanks to the OEIS (the On-Line Encyclopedia of Integer Sequences)

Relative validation of one generator w.r.t. another

Outline

Conclusion

- Generation of structured arrays
- Useful for automatically testing programs taking these arrays as inputs (bounded-exhaustive testing)
- Also shows how verification tools can facilitate the design and implementation of C programs enumerating combinatorial structures
- Library of structured array generators, formally specified and automatically verified
- Patterns of generation
- Perspectives: Proof of more efficient algorithms

Questions

- Thanks for your attention
- Questions?

References

Jörg Arndt.
Matters Computational - Ideas, Algorithms, Source Code [The fxtbook].
2010.
Published electronically at http://www.jjj.de.