J. Software Engineering & Applications, **2009, 2: 195-199**
doi:10.4236/jsea.2009.23026 Published Online October 2009 (http://www.SciRP.org/journal/jsea)

Formal Derivation of the Combinatorics Problems with *PAR* Method

Lingyu SUN[1], Yatian SUN[2]

[1]Department of Computer Science, Jinggangshan University, Ji'an, China; [2]School of Chemistry and Materials Science, University of Science and Technology of China, Hefei, China.
Email: sunlingyu@jgsu.edu.cn

Received May 12th, 2009; revised June 13th, 2009; accepted June 17th, 2009.

ABSTRACT

Partition-and-Recur (PAR) method is a simple and useful formal method. It can be used to design and testify algorithmic programs. In this paper, we propose that PAR method is an effective formal method on solving combinatorics problems. Furthermore, we formally derive combinatorics problems by PAR method, which cannot only simplify the process of algorithmic program's designing, but also improve its automatization, standardization and correctness. We develop algorithms for two typical combinatorics problems, the number of string scheme and the number of error permutation scheme. Lastly, we obtain accurate C++ programs which are transformed by automatic transforming system of PAR platform.

Keywords: *PAR Method, Formal Derivation, Combinatorics, Algorithmic Programs*

1. Introduction

Computer Science is a science of developing correct and efficient algorithms. Its main research object is discrete data handled by computer. As long as the algorithm involves iterant calculation, each traditional strategy of algorithm design can embody the principle of partition and recurrence. Partition is a general approach for dealing with complicated objects and is typically used in divide-and-conquer approach. Recurrence is used in algorithm analysis and dynamic programming approach. *PAR* method is a unified approach of developing efficient algorithmic programs and its key idea is partition and recurrence. Using *PAR* method, we begin from the formal specification of a specified problem, partition the problem into a couple of sub problems, and develop an efficient and correct algorithm represented by recurrence and initiation [1−3].

Combinatorics is a science of studying discrete objects. In general, it includes basic theories, counting methods and algorithms widely used in combination. Combinatorics algorithms are based on combinatorics and make people feel that computer may have its own thought.

However, many programming teaching materials can only present combinatorics algorithms but cannot provide the formal derived procedures that design the specific algorithms from the unresolved combinatorics problems. So algorithm designers cannot understand the essence of algorithms and cannot improve their capability of algorithm design [4].

PAR method embodies rich mathematic ideology, it provides many powerful tools and appropriate developing environment. We can avoid making a choice among various design methods by adopting *PAR* method to develop combinatory algorithms. It can also change many creative labor to mechanized labor, and can finally improve algorithmic programs design's automatization, standardization and correctness [5,6].

In this paper, we propose that *PAR* method is an effective formal method on solving combinatorics problems. We formally derive several typical algorithms of combinatorics problems by *PAR* method, which cannot only solve the above problems, but also can assure their correctness on logic. These combinatorics problem instances include the number of string scheme, the number of error permutation scheme, maximum summary [4], the longest common subsequence [4], minimum spanning tree [4,7] and the Knapsack problem [4], etc. Because of the limited space, we only describe the whole developing process of two specific combinatorics problem instances, the number of string scheme and the number of error permutation scheme, which are derived by *PAR* methods.

2. The Developing Steps of the Number of String Scheme Derived by *PAR* Methods

Suppose that A=$\{a,b,c,d\}$, n is given, and we want to select *n* elements to compose string, in which element *a* and *b* cannot be adjacent elements. How many schemes are there in a string with *n* elements? [4]

2.1 The Formal Function Specification of the Number of String Scheme

PQ: Given integer *n*, set $A=\{a,b,c,d\}$, string *S* stores *n* elements of set *A*.

PR: Let $String(n)=\{S \mid S[k]=a \vee S[k]=b \vee S[k]=c \vee S[k]=d \wedge 1 \le k < n\}$ and $F(S,i)=\forall(j: i \le j < n : S[j..j+1] \ne ab \wedge S[j..j+1] \ne ba)$, then

$$
\begin{aligned}
Z_n &= N(S : S \in String(n) : \\
&\quad \forall(j : 1 \le j < n : S[j \ldots j+1] \ne ab \wedge S[j \ldots j+1] \ne ba)) \\
&= \{\textit{According to the definition of } F(S,i)\} \\
&\quad N(S : S \in String(n) : F(S,1)) \\
&= \{\textit{equivalent transformation of quantifier}\} \\
&\quad \sum(S : S \in String(n) \wedge F(S,1) : 1) \qquad (1)
\end{aligned}
$$

2.2 Partition the Problem Based on the Post-Assertion

We partition computing Z_n into computing X_n and Y_n, each of that has the same structure with Z_n.

$$
\begin{aligned}
X_n &= \sum(S : S \in String(n) \wedge F(S,1) \\
&\quad \wedge (S[1]=a \vee S[1]=b) : 1) \qquad (2)
\end{aligned}
$$

In Equation (2), X_n denote the number of *n* elements' string scheme that element *a* and *b* cannot be adjacent elements and the initial element is *a* or *b*.

$$
\begin{aligned}
Y_n &= \sum(S : S \in String(n) \wedge F(S,1) \\
&\quad \wedge (S[1]=c \vee S[1]=d) : 1) \qquad (3)
\end{aligned}
$$

In Equation (3), Y_n denote the number of *n* elements' string scheme that element *a* and *b* cannot be adjacent elements and the initial element is *c* or *d*.

$$
\begin{aligned}
Z_n &= \sum(S : S \in String(n) \wedge F(S,1) : 1) \\
&= \sum(S : S \in String(n) \wedge F(S,1) \\
&\quad \wedge (S[1]=a \vee S[1]=b \vee S[1]=c \vee S[1]=d) : 1) \\
&= \{\textit{Range Disjunction}\} \\
&\sum(S : S \in String(n) \wedge F(S,1) \wedge (S[1]=a \vee S[1]=b) : 1) + \\
&\sum(S : S \in String(n) \wedge F(S,1) \wedge (S[1]=c \vee S[1]=d) : 1) \\
&= \{\textit{According to the definition of } X_n \textit{ and } Y_n\} \\
&\quad X_n + Y_n \qquad (4)
\end{aligned}
$$

According to the equation (4), we can partition computing Z_n into computing X_n and Y_n.

2.3 Construct the Recurrence Relation

Suppose X_{n-1} denotes the number of *n-1* elements' string scheme that satisfy the condition and the initial element is *a* or *b*. Y_{n-1} denotes the number of *n-1* elements' string scheme that satisfy the condition and the initial element is *c* or *d*.

$$
\begin{aligned}
X_n &= \sum(S : S \in String(n) \wedge F(S,1) \\
&\quad \wedge (S[1]=a \vee S[1]=b) : 1) \\
&= \sum(S : S \in String(n) \wedge F(S,1) \wedge (S[1] \\
&= a \vee S[1]=b) \wedge (S[2] \\
&= a \vee S[2]=b \vee S[2]=c \vee S[2]=d) : 1) \\
&= \{\textit{Range Disjunction}\} \\
&\sum(S : S \in String(n) \wedge F(S,1) \wedge (S[1]=S[2]) \\
&\wedge (S[2]=a \vee S[2]=b) : 1) + \\
&\sum(S : S \in String(n) \wedge F(S,1) \wedge (S[1]=a) \\
&\wedge (S[2]=c \vee S[2]=d) : 1) + \\
&\sum(S : S \in String(n) \wedge F(S,1) \wedge (S[1]=b) \\
&\wedge (S[2]=c \vee S[2]=d) : 1) \\
&= \{\textit{According to the definition of } F(S,i)\} \\
&\sum(S : S \in String(n) \wedge (S[1]=S[2]) \wedge F(S,2) \\
&\wedge (S[2]=a \vee S[2]=b) : 1) + \\
&\sum(S : S \in String(n) \wedge (S[1]=a) \wedge F(S,2) \\
&\wedge (S[2]=c \vee S[2]=d) : 1) + \\
&\sum(S : S \in String(n) \wedge (S[1]=b) \wedge F(S,2) \\
&\wedge (S[2]=c \vee S[2]=d) : 1) \\
&= \{\textit{According to the definition of } X_{n-1} \textit{ and } Y_{n-1}\} \\
&\quad X_{n-1} + 2 \times Y_{n-1} \qquad (5)
\end{aligned}
$$

According to the equation (5), we can partition computing X_n into computing X_{n-1} and $2 \times Y_{n-1}$.

$$
\begin{aligned}
Y_n &= \sum(S : S \in String(n) \wedge F(S,1) \\
&\quad \wedge (S[1]=c \vee S[1]=d) : 1) \\
&= \sum(S : S \in String(n) \wedge F(S,1) \wedge (S[1] \\
&= c \vee S[1]=d) \wedge (S[2]=a \vee S[2] \\
&= b \vee S[2]=c \vee S[2]=d) : 1)
\end{aligned}
$$

= {*Range Disjunction*}

$$\sum (S : S \in String(n) \wedge F(S,1) \wedge (S[1] = c \vee S[2] = d) \wedge (S[2] = a \vee S[2] = b) : 1) +$$

$$\sum (S : S \in String(n) \wedge F(S,1) \wedge (S[1] = c \vee S[2] = d) \wedge (S[2] = c \vee S[2] = d) : 1)$$

= {*Range Disjunction*}

$$\sum (S : S \in String(n) \wedge F(S,1) \wedge (S[1] = c) \wedge (S[2] = a \vee S[2] = b) : 1) +$$

$$\sum (S : S \in String(n) \wedge F(S,1) \wedge (S[1] = d) \wedge (S[2] = a \vee S[2] = b) : 1) +$$

$$\sum (S : S \in String(n) \wedge F(S,1) \wedge (S[1] = c) \wedge (S[2] = c \vee S[2] = d) : 1) +$$

$$\sum (S : S \in String(n) \wedge F(S,1) \wedge (S[1] = d) \wedge (S[2] = c \vee S[2] = d) : 1)$$

= {*According to the definition of* $F(S,i)$ }

$$\sum (S : S \in String(n) \wedge (S[1] = c) \wedge F(S,2) \wedge (S[2] = a \vee S[2] = b) : 1) +$$

$$\sum (S : S \in String(n) \wedge (S[1] = d) \wedge F(S,2) \wedge (S[2] = a \vee S[2] = b) : 1) +$$

$$\sum (S : S \in String(n) \wedge (S[1] = c) \wedge F(S,2) \wedge (S[2] = c \vee S[2] = d) : 1) +$$

$$\sum (S : S \in String(n) \wedge (S[1] = d) \wedge F(S,2) \wedge (S[2] = c \vee S[2] = d) : 1)$$

= {*According to the definition of* X_{n-1} *and* Y_{n-1} }

$$2 \times X_{n-1} + 2 \times Y_{n-1} \qquad (6)$$

According to the equation (6), we can also partition computing Y_n into computing $2\times X_{n-1}$ and $2\times Y_{n-1}$.

2.4 Developing Loop Invariant

Suppose variant *x* stores the value of X_i, variant *u* stores the value of X_{i+1}, variant *y* stores the value of Y_i, variant *v* stores the value of Y_{i+1}, variant *z* stores the value of Z_i, where $X_1=2$, $Y_1=2$, $Z_1=4$.

LI: $x = X_i \wedge u = X_{i+1} \wedge y = Y_i \wedge v = Y_{i+1} \wedge z = Z_i \wedge (1 \le i \le n)$

2.5 Developing Corresponding *RADL* Program

The Recurrence-based Algorithm Design Language (*RADL*) program of the number of string scheme, derived by *PAR* methods, is shown in Algorithm 1. By the automatic program transforming system of *PAR* platform, we can get the Abstract Programming Language (*APLA*) program of the number of string scheme which is transformed from the *RADL* program and is shown in Algorithm 2. Finally, we transform the *APLA* program of the number of string scheme to *C++* program, which can get accurate running result.

Algorithm 1 (*The RADL program of string scheme*)

```
|[in n:integer; i:integer; x,y,u,v: integer; out z:integer;]|
{PQ∧PR}
Begin: i=1++1; x=2; y=2; z=4;
A_I:  $x = x(i) \wedge u = x(i+1) \wedge y = y(i) \wedge$
      $v = y(i+1) \wedge z = z(i) \wedge (1 \le i \le n)$
Termination:   i=n;
Recur:         u=x+2*y;   v=2*x+2*y;
z=u+v; x=u; y=v;
End.
```

Algorithm 2 (*The APLA program of string scheme*)

```
var: n:integer; i:integer; x,y,u,v: integer; z:integer;
begin:
  write("Please input integer n value");
  read(n);
  i:=1;x:=2;y:=2;z:=4;
  do (¬(i=n)) →
      u:=x+2*y;
      v:=2*x+2*y;
      z:=u+v
      x:=u;
      y:=v;
      i:=i+1;
  od
  write("The Number of string scheme:", z);
end.
```

3. The Developing Steps of the Number of Error Permutation Scheme Derived by *PAR* Methods

Suppose that the original arrange scheme is $A = [a_1 \cdots a_i \cdots a_n]$, which is satisfied with each element is different. We want to interlace *n* elements of original permutation scheme *A* to compose new permutation scheme *B*, in which each element cannot be the same position in *A*. How many schemes are there in error permutation with *n* elements? [4]

3.1 The Formal Function Specification of the Number of Error Permutation Scheme

PQ: Given the original permutation scheme $A = [a_1 \cdots a_i \cdots a_n]$, which is satisfied with $\forall(i, j :$

$(1 \le i < j \le n) : (A[i] \ne A[j]))$.

PR: Let $Perm(A) = \{B \mid B[j] = A[i] \wedge B[j] \ne B[k] \wedge 1 \le i, j, k \le n \wedge j \ne k\}$, then

$$D_n = N\big(B : B \in Perm(A) \wedge \forall(j : 1 \le j \le n : B[j] \ne A[j])\big)$$
$$= \{\textit{equivalent transformation of quantifier}\}$$
$$\sum\big(B : B \in Perm(A) \wedge \forall(j : 1 \le j \le n : B[j] \ne A[j]) : 1\big)$$
$$= \{\exists'(i : 1 \le i < n : B[n] = A[i]) = true\}$$
$$\sum\big(B : B \in Perm(A) \wedge \forall(j : 1 \le j \le n : B[j] \ne A[j]) \wedge \exists'(i : 1 \le i \le n : B[n] = A[i]) : 1\big) \quad (7)$$

3.2 Partition the Problem Based on the Post-Assertion

We partition computing D_n into computing U_n and V_n, each of that has the same structure with D_n.

$$U_n = \sum\big(B : B \in Perm(A) \wedge \forall(j : 1 \le j \le n : B[j] \ne A[j]) \wedge \exists'(i : 1 \le i < n : B[n] = A[i] \wedge B[i] = A[n]) : 1\big) \quad (8)$$

In Equation (8), U_n denotes the number of n elements' error permutation scheme in which a_n must be the i^{th} element in new permutation B.

$$V_n = \sum\big(B : B \in Perm(A) \wedge \forall(j : 1 \le j \le n : B[j] \ne A[j]) \wedge \exists'(i : 1 \le i < n : B[n] = A[i] \wedge B[i] \ne A[n]) : 1\big) \quad (9)$$

In Equation (9), V_n denotes the number of n elements' error permutation scheme in which a_n cannot be the i^{th} element in new permutation B.

$$D_n = \sum\big(B : B \in Perm(A) \wedge \forall(j : 1 \le j \le n : B[j] \ne A[j]) \wedge \exists'(i : 1 \le i < n : B[n] = A[i]) : 1\big)$$
$$= \{\textit{Range Disjunction}\}$$
$$\sum\big(B : B \in Perm(A) \wedge \forall(j : 1 \le j \le n : B[j] \ne A[j]) \wedge \exists'(i : 1 \le i < n : B[n] = A[i] \wedge B[i] = A[n]) : 1\big) +$$
$$\sum\big(B : B \in Perm(A) \wedge \forall(j : 1 \le j \le n : B[j] \ne A[j]) \wedge \exists'(i : 1 \le i < n : B[n] = A[i] \wedge B[i] \ne A[n]) : 1\big)$$
$$= \{\textit{According to the definition of } U_n \textit{ and } V_n\}$$
$$U_n + V_n \quad (10)$$

According to the equation (10), we can partition computing D_n into computing U_n and V_n.

3.3 Construct the Recurrence Relation

Suppose D_{n-1} denotes the number of *n-1* elements' error permutation scheme, D_{n-2} denotes the number of *n-2* elements' error permutation scheme.

$$U_n = \sum\big(B : B \in Perm(A) \wedge \forall(j : 1 \le j \le n : B[j] \ne A[j]) \wedge \exists'(i : 1 \le i < n : B[n] = A[i] \wedge B[i] = A[n]) : 1\big)$$
$$= \{\textit{Generalized Range Disjunction}\}$$
$$\sum\big(i : 1 \le i < n : \sum\big(B : B \in Perm(A) \wedge \forall(j : 1 \le j \le n : B[j] \ne A[j]) \wedge B[n] = A[i] \wedge B[i] = A[n] : 1\big)\big)$$
$$= \sum\big(i : 1 \le i < n : \sum\big(B : B \in Perm(A) \wedge \forall(j : 1 \le j < i \vee i < j \le n : B[j] \ne A[j]) : 1\big)\big)$$
$$= \{\textit{According to the initial definition of } D_n\}$$
$$\sum(i : 1 \le i < n : D_{n-2})$$
$$= (n-1) \times D_{n-2} \quad (11)$$

According to the equation (11), we can partition computing U_n into computing $(n-1) \times D_{n-2}$.

$$V_n = \sum\big(B : B \in Perm(A) \wedge \forall(j : 1 \le j \le n : B[j] \ne A[j]) \wedge \exists'(i : 1 \le i < n : B[n] = A[i] \wedge B[i] \ne A[n]) : 1\big)$$
$$= \{\textit{Generalized Range Disjunction}\}$$
$$\sum\big(i : 1 \le i < n : \sum\big(B : B \in Perm(A) \wedge \forall(j : 1 \le j \le n : B[j] \ne A[j]) \wedge B[n] = A[i] \wedge B[i] \ne A[n] : 1\big)\big)$$
$$= \sum\big(i : 1 \le i < n : \sum\big(B : B \in Perm(A) \wedge \forall(j : 1 \le j < n : B[j] \ne A[j]) : 1\big)\big)$$
$$= \{\textit{According to the initial definition of } D_n\}$$
$$\sum(i : 1 \le i < n : D_{n-1})$$
$$= (n-1) \times D_{n-1} \quad (12)$$

According to the equation (12), we can partition computing V_n into computing $(n-1) \times D_{n-1}$.

According to the equation (10), (11), (12), we have the recurrence: $D_n = U_n + V_n = (n-1) \times (D_{n-1} + D_{n-2})$, where $D_1=0$, $D_2=1$. That is to say, we can partition computing D_n into computing $(n-1) \times D_{n-1}$ and $(n-1) \times D_{n-2}$.

3.4 Developing Loop Invariant

Suppose variant *s1* stores the value of *d*(*i-2*), variant *s2* stores the value of $d(i-1)$, variant d stores the value of $d(i)$, where $D_1=0$, $D_2=1$.

LI: $s1 = d(i-2) \wedge s2 = d(i-1) \wedge d = d(i) \wedge (1 \le i \le n)$

3.5 Developing Corresponding *RADL* Program

The *RADL* program of the number of error permutation scheme, derived by the *PAR* methods, is shown in Algorithm 3. By the automatic program transforming system of *PAR* platform, we can get the *APLA* program of the number of error permutation scheme which is transformed from the *RADL* program and is shown in Algorithm 4. Finally, we transform the *APLA* program of the number of error permutation scheme to *C++* program, which can get accurate running result.

Algorithm 3 (*The RADL program of error permutation scheme*)

```
|[in n:integer; i,s1,s2: integer; out d:integer;]|
{PQ∧PR}
Begin: i=3++1; s1=0; s2=1;
A_I: $s1 = d(i-2) \wedge s2 = d(i-1) \wedge d = d(i) \wedge (3 \le i \le n)$
Termination: i=n+1;
Recur:     d= (i-1)*(s1+s2);
s1=s2;s2=d;
End.
```

Algorithm 4 (*The APLA program of error permutation scheme*)

```
var: n:integer; i:integer; s1,s2: integer; d:integer;
begin:
  write("Please input integer n value");
  read(n);
  i:=2;s1:=0;s2:=1;
  do (¬(i=n)) →
        d= (i-1)*(s1+s2);
        s1:=s2;
        s2:=d;
        i:=i+1;
  od
  write("The Interlaced Arrange Scheme:", z);
end.
```

4. Conclusions

We developed algorithmic programs for the number of string scheme and the number of error permutation scheme which are typical combinatorics problems. It is revealed that PAR method has particular merit. It is apt to understand and demonstrate the ingenuity and correctness of an algorithm by formula deduction. Compared with other derivation, our algorithmic programs are more precise and simple than the representation of algorithm in natural language, flowchart and program. It also shows that using *PAR* method in developing combinatorics problem is a very natural meaningful research work. This can not only expand the application of *PAR* method, but also prove that the *PAR* method is a unified and effective approach on solving combinatorics problems.

5. Acknowledgment

This paper is supported by the international cooperation project of Ministry of Science and Technology of PR China, grant No. CB 7-2-01, and by Science and Technology research project of Jiangxi Municipal Education Commission under Grant No. GJJ09590.

REFERENCES

[1] J. Y. Xue, "A unified approach for developing efficient algorithmic programs [J]," Journal of computer Science and Technology, Vol. 12, No. 4, pp. 103–118, 1997.

[2] J. Y. Xue, "Formal derivation of graph algorithmic programs using Partition-and-Recur [J]," Journal of Computer Sciences and Technology, Vol. 13, No. 6, pp. 95–102, 1998.

[3] J. Y. Xue, "A practicable approach for formal development of algorithmic programs [C]," The Proceedings of The international Symposium on Future software Technology (ISFS'99), Published by Software Engineers Associations of Japan, pp. 212–217, 1999.

[4] L. Y. Sun, "The applied research of PAR method on combinatorics problems [D]," Jiangxi Normal University, 2007.

[5] J. Y. Xue, "Research on formal development of algorithmic program [J]," Journal of Yunnan University (natural sciences), Vol. 19, pp. 283–288, 1997.

[6] Y. Q. Li, "Partition-and-recur method and its applications [J]," Computer Engineering and Applications, Vo1. 27, No. 11, pp. 77–79, 2000.

[7] L. Y. Sun and J. Y. Xue, "Formal derivation of the minimum spanning tree algorithm with PAR Method [J]," Computer Engineering, Vo1. 32, No. 21, pp. 85–87, 2007.