计算机科学 1995 Vol. 22№. 2

一种基于算法骨架的函数式程序变换技术

姚春玲　孙永强
（上海交通大学计算机系　上海 200030）

摘　要　Program transformation techniques have been used as one of the main approaches of improving program execution efficiency. The implicit parallelisms of functional program make it suitable to process the derivation for parallel algorithms. Algorithmic skeletons, mentioned in this paper, are essentially high-order functions which express for group of problems the general algorithmic models as well as computation structures. We employ several algorithmic skeletons which are already at hand, make some choices of the suitable ones to describe the problem specification under the guidance of the cost performance attached with each algorithmic skeleton, and supply the compositions between them. The transformation mechanism, built between skeletons, guarantees the portability between different implementation structures. And the conversion from skeleton to its specific corresponding implementation structures furnishes us with the parallel and efficient implementation version for the problem being soloved.

关键词　Algorithmic skeleton, Program transformation, Performance model, Data distribution.

一、动机及背景

如何提高函数式程序设计语言在传统冯·诺依曼机器上的执行速度及效率，一直是该领域中研究的主要论题，对此，并行图归约技术、并行闭包归约、并行编译、并行程序转换等等技术相继成为改善这种状况的措施。

将用户程序经过一系列保正确的变换（进行算法及数据结构的求精）至可在一特定的并行环境下运行的并行程序，最大限度地利用这一并行结构的优势来获取用户程序的高效执行，便是程序转换的任务。函数式语言具有隐涵的并行性，较传统的命令式语言更适合于并行性的识别及开发。随着机器结构的发展及完善，各种并行结构的出现为函数式语言的并行实现提供了一条现实的前景。

程序变换方法大体上可分纵向变换和横向变换两大类：纵向变换是指将某一级别的程序转换成较低抽象级别的程序，如 Bauer 主持的 CIP2 项目和徐家福主持的 NDAUTO 系统；而横向变换则是指在相似的抽象级别上将一个程序变换为一个与之等价但执行效率更高的程序。将递归函数式程序变换成语义上等价的迭代程序文本是横向程序变换的一个重要方式。

横向变换的研究可追溯到 1966 年的 Copper 变换，而爱丁堡大学的 Burstall 和 Darlington 的研究以及他们所研制的 ZAP 系统是目前这方面最有代表性的工作[7]。由于程序的可读性与程序的效率之间的矛盾是在程序结构中，尤其是在递归或循环结构中[7]，所以针对这一问题的解决方法是消除递归，将其转换至更高效的形式，故 Darlington 的系统也可称为是一种递归程序的转换系统。该系统由一些简单的转换规则（如 Definition, Instantiation, Unfold, Fold, Abstraction, Law）及其应用这些规则的策略组成[6]

二、算法骨架方法

1. 产生的背景

算法骨架首先是由 Cole 针对并行机器的高级程序设计方法而提出的概念，其研究方法的基础是识别出算法技术的一个相似集合，每个算法技术都最适合于某一特定范围的问题，对其它范围的问题则不适应。每一集合都对应一个适合于并行实现的计算结构，一旦关键的计算部分被并行有效地执行，那么对这一算法技术的任一应用都可受益。我们将

姚春玲　博士研究生，主要从事并行处理，并行算法推导及函数式程序转换等方向的研究。孙永强　教授，博士生导师。

这些算法技术嵌入到程序的句法结构中，程序设计者就可以采用其中之一类作为每一个程序的中心结构。

借助于一个连续的高阶函数方法来描述一类算法，高阶函数只指明了该算法类的更高一级上的计算结构，而不关心具体问题的低级情况。对一具体问题的求解是将现有的高阶函数作用在具体的函数上来实现的，也即具体的函数是高阶函数的一个自变量。

2. 算法骨架及其性质

1)算法骨架在形式上是一个高阶函数，被用来作为程序最外层函数的形式[1]。在我们所要描述的程序设计模型中，提供给程序设计者一个由特定的高阶函数(骨架)组成的选集(selection)。每一个骨架描述的是一类相似算法的关键计算结构，表示了一类应用程序的通用的算法形式，而没有过多地指明细节情况。从选集中选择一个来作为程序的最外层函数(f)，程序设计者可以利用所使用的语言定义一个函数(g)，使得从该选集中选择的高阶函数作用在用户定义的函数上之后，产生程序的最终解 fg。

只有当一个程序的最外层函数匹配了其中一个可以被接受的高阶函数(骨架)时，则该程序称为是合法的。获得了这种程序结构上的约束后，我们就可以不必关心较低一级的程序细节，因为这些细节不会影响到算法的关键结构或其实现，程序的较低一级上的细节描述则全部依赖宿主语言。

程序中所有的并行性均来自于它所拥有的骨架的特性，而骨架所作用的函数则可以被串行或并行执行。

2)每个骨架都有一类机器结构集合与之对应，在该集合中存在骨架的有效的并行实现方案，而骨架的一个最优实现结构则是由其性能模型来指明。由于骨架的含义独立于它的任何具体实现，所以骨架程序既可以在串行机上高效执行，又可以在不同并行机间进行移植。总的来讲，借助一种转换机制任何骨架都可以在任一机器上执行。

3)由于骨架被定义成一些特殊类型的函数，所以它们可以用其它的函数(尤其是骨架)来定义或实现。这就体现了骨架的一种层次关系，如图 1，S 表示骨架，Aj 代表结构。这种层次结构可用于两个不同的目的：图中不同层次上的骨架对应了不同级别的抽象；同时也表示了一种转换关系，即使它们是在同一抽象级别上。所以骨架又可以用作为针对并行程序开发的一种基于转换的模型[2]，这种模型的并行实现的关键是保证每个合法的高阶函数所描述的计算结构可以得到并行的实现，换句话说，该模型对应一个很好的并行算法技术。

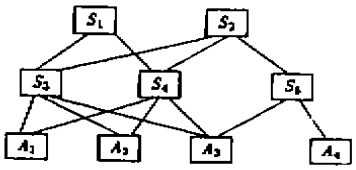

图 1

4)骨架的结构中隐含着对问题的并行分解[4]，因此在编程者选择了一个骨架的同时，也选择了一种问题的并行分解策略。

总而言之，对程序设计者来说，每个骨架是一个高阶函数，或者是一个程序样板(如果基本语言是命令式语言)，它是一个算法工具集合，从中一定可以构造出一个具体问题程序；对系统实现者来说，每个骨架是一个通用的计算模型，和它等价的高效的并行实现结构同时也要求是定义好的，每个骨架对每个并行机器要求有不同的实现方案，这些不同对程序设计者是不可见的，他所见到的是独立于程序设计框架的相同的机器。

在系统的最后实现上，数据及编译好的用户代码及被选中的将要被执行的预定义的骨架系统代码一起被装载到并行硬件结构上，硬件执行骨架，调用用户进程，返回执行结果，这些过程对用户隐蔽。

三、基于算法骨架的函数式程序转换方法

这种变换方法采用骨架作为函数式程序并行实现的基本建筑模块，通过提供在骨架之间的转换机制以达到不同机器结构间的兼容。每个骨架/机器序对的资源分配由它所对应的性能模型来指导，是在实现过程中以交互方式获得的。

1. 几种最基本的常用算法骨架

PIPE 骨架：描述的是简单的线性进程的并行性，一些函数组合在一起处理数据流，该骨架的并行性的获得是将每一个函数放在一个不同的处理机上执行，这一思想可以很容易地扩展到更高一阶的情况上。其形式描述为：

pipe::[x→x]→[x→x]

pipe=fodr1( )

DC 骨架：许多算法都采用这一种形式，将一个

大的任务分解成若干子任务，然后独立地求解，最后将它们的结果结合起来。这种方式称为 Divide-and Conquer(分治)模式，也即 DC 骨架。小任务(t)可直接由本地处理器求解，而大的任务则被分成(d)子任务，被传送到其它处理器上递归实现，子结果的组合(c)就产生了最终解。其形式描述为：

```
DC::(x→BOOL)→(x→y)→(x→[x])→([y]→y)→x→y
DC t s d c x |t x=s x
             |not(tx)=(c. map(DC t s d c).d)x
```

FARM 骨架，所表示的是最简单的数据并行性。函数的作用对象是一个任务序列(list of jobs)，该函数还要求有一个环境用以表示所有任务的公共数据，通过采用设置重复处理器处理任务的方式来获得并行性。其形式描述为：

```
FARM::(x→y→z)→x→([y]→[z])
FARM f env=map. (f. env)
```

RaMP(Reduce and Map-over-Pairs)骨架：这一算法类描述的情况是系统中的每个对象都可以和其它对象相互作用，每单个作用的结果组合在一起产生最终结果。该骨架的使用主要针对初始规范，将它从一种形式转换到另一形式，例如为每个对象实施计算操作，或者将 PIPELINE 作用在函数 f,g 上。其形式描述为：

```
RaMP::(X→x→y)→(y→y→y)→[x]→[y]
RaMP f g xs=map h xs
where h x =folder1 g(map(f x )xs)
```

DMPA (Dynamic-Message-Passing-Architecture)骨架：该算法描述的是更为动态的情形。任一进程都可以通过信息传输和其它任何进程发生相互作用，并由运行时间数据(run time data)来决定。每一进程都有一内部状态记录进程的局部值，来自其它进程的信息会改变该进程的状态并产生新的信息，这种骨架的并行性来自于不同处理机上对同一进程的求值。

```
DMPA::{{x}→{(int,x)}}→{(int,x)}→{x}
DMPA (Pi initStatei | 1<=i<=n} initMess
  =filterms 0 mess
   where mess=Pi initState1(filterms 1 mess)∪
…∪  Pn initStaten(filterms n mess)∪ initMess
   filterms i ms-{conts |(j,conts) ms,i==j}
   Pi localState(c ∪ cs)=replies ∪ Pi updState
cs[1]
```

以上所有的骨架对应的都是 MIMD 模型。

2. 性能模型(performance models)

每一个骨架/机器序对都有一个性能模型与之对应，用以预测用骨架书写的运行在该机上的程序的性能。编程者、转换系统及编译器根据这一性能模型来指导程序开发过程中各级上的决策生成，资源分配尤其要依赖这些性能模型。

对并行程序性能的预测是十分困难的，目前的解决方案是"性能跟踪"策略。编好的程序，执行后观察其性能，得出的结果再用来修改该程序中的资源分配策略，修改后的程序再次执行，以视其性能是否有了提高。这种方法仿佛夜行，因为编程者不清楚改善程序性能的关键所在。另一种是采用"性能模型"的方法，给定一个程序，该模型可以预测出程序的性能，并建议以何种措施来改善性能。这样的一个模型要求一个分析公式的集合，分析公式要求以程序及其机器结构作为参数。不幸的是，工作量及其复杂度将是十分巨大的，我们清楚，没有一种现实方法来对可执行在任意机器上的程序进行性能的预测。

在此，我们将程序看作为骨架的示例，每一骨架对应于一类机器结构，在这种具体情况下进行并行程序性能的预测将是较为现实的方法。

与每个骨架/机器序对相对应的性能模型是在程序设计过程中逐渐建立起来的，首先建立一个初始模型，经实验得到检验及完善，一直修改到可以可靠地预测性能时为止。这种方法与前所提及的"性能跟踪"方法有相似之处。

考虑一个简单的例子，D-C 骨架，它对应的实现结构是分布式存储器机器结构。这类机器的存储器存取时间随机，局部存取代价小于远地存取，影响性能的两个重要因素是进程的粒度及数据分配问题，模型还要考虑每个变量函数的复杂度及处理器间通讯速度，将这些因素都考虑进去，这个 D-C 应用事例可以并行地得到处理，只要满足下列条件：

$$TsolG > TdivG + TsolG/2 + TcombG/2 + Tcomms$$

其中 Tsolx：在一个处理器上求解大小为 x 的问题需要的时间；Tdivx：将大小为 x 的问题划分成两个子问题的时间；Tcombx：将大小为 x 的两个子问题结合起来的时间；Tcomms：处理器间问题及结果通讯的时间。可以将问题扩充到大小为 G、处理器数目为 M 的情况，则条件变为：

$$TsolG = \sum_{i=1}^{\log M} (TdivG/2 + TsolG/2 + TcombG/2 + Tcomms) + TsolG/M$$

求解这个方程中M的值即是处理器的最佳数目。另外，有关共享数据的处理问题也应该考虑进去，是否应该只计算一次，但要求远程访问，还是应该计值一次，但要拷贝到每个处理器上去，还是要求每个处理器对共享变量重新计值。

在某些情况下，一些骨架在有些机器上存在多种实现方案，那么就根据性能模型选择一个优化的，这种性能模型方法对并行程序的转换及并行程序设计将是十分有用的。

3. 程序转换

程序转换存在于程序开发的所有过程中，在最高一级上，将高级问题规范形式转换至初始骨架形式；在低一级上，将程序从一个骨架转换到另一骨架（为了实现移植）；在最低一级上，用以将一个针对特定机器结构的程序调整到特定的机器结构上。

程序转换为程序的移植提供了一种较自然的手段，如果一个以一种骨架书写的程序，无法影射到一给定的结构上实现，那么将它重写成另外一种骨架形式，能够在这一给定的结构上有效地运行，这种过程就实现了一种程序级的转换。

1）骨架之间的转换。例如：将RaMP在PIPELINE上实现。一个借助于RaMP骨架书写的程序在长度为xs－2的流水线上实现如下：

```
RaMP f g xs = (map and PIPE (map map (map g xs))
      map(pair unit))xs
   where g b(a,c)= (a,g(f a b )c)
     pair a b=(b,a)
```

另外，又可以以FARM形式在一个分布式的结构中实现：

```
RaMP f g xs=FARM h (f,g,xs)xs
where h(f,g,xs)x=foldrl g(map(f x)xs)[1]
```

因此我们借助于程序转换可以把一可移植的高级规范影射到当前所具备的任何结构上去，并且可调整一规范的示例以充分地利用机器结构的性能。

2）骨架到硬件结构的转换。在这一级转换中，主要针对两类机器结构：静态结构和动态信息传输结构DMPA (dynamic-message-passing-architecture)。前者其内部处理机的互联固定，可以被函数语言直接表达出来，函数是处理机的模型，函数的组合操作则是处理机内部通讯结构的模型。例如(f。g)x其意义为f(g(x))，表示计算f的处理器接受的是计算g(x)的处理器的结果。

动态信息传输结构(DMPA)具有一种信息导向机制，导引MSG(目的，内容)形式信息到目的处理器上，因此，任何处理器都可以向其它处理器发送信息，函数被作为进程的模型，信息导向则是由集合来表示的。这种导引机制的实现在ALICE机上采用的是DELTA网络转换开关，而在Thinking Machines Connection Machine CM2上，导引是通过超立方体硬件及导引软件来实现的。

由函数式语言来描述这两类结构如图2和图3所示。

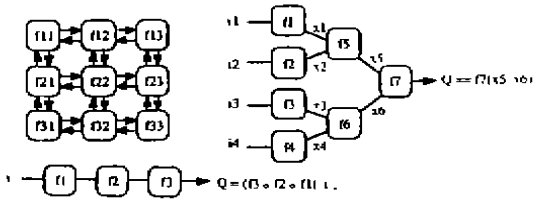

图2　静态结构

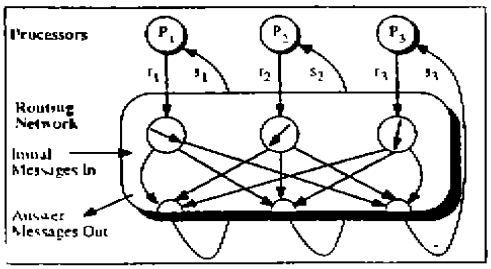

图3　动态结构

对于静态结构所采用的转换方法是对一问题规范进行推导转换，直到它满足一种函数结构为止，该函数结构要求相似于静态的目标机器结构。与机器结构中的处理器相对应的功能函数则被抽象并编译成机器代码加载到目标结构上实现。

动态结构中问题规范描述的是针对某一输入信息其相应的输出信息的情况，即要求有什么样的信息输出。向动态结构的转换即是冗余计算的消除过程（借助附加的信息传输），如果一处理器需要的中间结果是由另外处理器来计算的，则该处理器直接以信息传输的方式来获得而不是局部地进行信息的重新计算。当规范中对全局数据的依赖均已经转换而消除了之后，就获得了一高效率的算法，向动态结构的转换终止。下面以排序算法为例说明之。

·向静态结构的转换。设欲排序的序列为X，

$(X_1,\cdots X_n)$，X_i 在已排好序的序列中的位置是 1 加上所有比它小的元素的个数，及在未排列前所有在它左部的与之相等元素的个数。函数 posn(a,list)返回的值为元素 a 在 list 中的位置。

$$posn(X_i,sort[X_1,\cdots,X_n])=1+\#\{Xi<Xj|1<=i<=n\}+\#\{Xi=Xj|i<=1<=j\}$$

显然这个问题的规范与 PIPE 骨架相匹配，求解可以借助于流水线以 O(n)的时间来实现，而较优化的算法则需要 O(logn)的时间。显然我们有以下针对具体实现结构的问题规范：

(fO(logn),…,f2,f1)X=sort X

我们的任务就是推导出 f1,…,fO(logn)这些函数。

fO(logn),…,f2,f1 X=[(Xj|posn(Xj,sort X)=1),
…
(Xj|posn(Xj,sort X)=n)]

在流水线的最后一步上，也即最小元素被求出之前，流水线中的情形如下：

$Xi<Xj<Xk\ 1<=i,j,k<=n,i<>j<>k$

和

$Xp<Xq<Xr\ 1<=p,q,r<=n,p<>q<>r$

那末，最小的元素只需一步就可确定下来，是 Xi 或者是 Xp。假设是 Xi，则第二小的元素或是 Xp 或是 Xj。显然我们在合并这两个排序表。很清楚，排序的流水线规范给我们以启示，这种流水线结构的问题求解也适合于用 mergesort 结构来求解。假设已存在一个 mergsort 结构，将它影射到 PIPELINE 上去，以获得我们所需的流水线上的各级功能函数 Pipestage(借助类型转换，在此从略)。

经这种转化而得到的流水线上一个长度为 n 的待排序表需要 O(log n)个处理器经 O(n)时间就可排好序，这种合成的流水线显然是优化的，如图 4 所示。

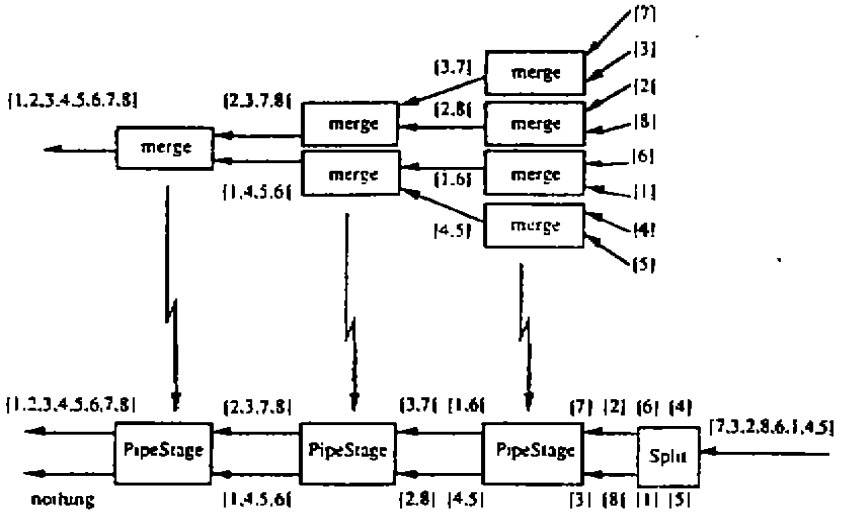

图 4

·向动态信息传输系统的转换。相比之下情况就比较复杂，这种转换是通过推导出处理器间交换信息的集合来实现的。进程对信息的获得是直接通过信息交换而不是对所需的信息的重新计算，转换过程中引入了一些规则，这些规则指明了哪一信息的出现能引起哪一些信息的产生，第一条规则指明了启动计算操作的初始信息及所产生的相应信息。这一方面的研究尚有许多工作要进行。

四、若干并行程序转换方法之比较

帝国大学 Darlington 领导的研究组正进行着一种基于骨架的函数式并行程序设计方面的研究，他们的骨架(如上面所列)被定义成带有特殊的并行实现策略的高阶函数，所采用的方法是基于等价骨架之间的转换，由每个骨架相对应的代价测度来指导转换，但缺陷是没有对数据划分问题给出明确的解答，而且只允许对最外层的函数实施并行，内层被调

用函数则要求串行执行，不支持骨架的组合操作。

另一个不同的但相关的研究方法是比萨大学P3L计划的一部分。P3L提供了一个骨架的基本集合和一个结构化的并行语言来支持基本骨架的结合操作，避免了骨架的组合问题；耶鲁大学的M. Chen和Y. Choo所进行的研究是一种采用特殊数据分配（称为index域）的针对矩阵算法的演算，在这些数据分布上的算法是由一些函数（从index域到其值域间的影射）来说明的，而不是依靠骨架。这些函数被称为是数据域（datafield），各种并行算法间的转换采用与不同数据域相关的代数规则来表达[3]。

柏林工大的Pepper研究组所作的工作主要也是并行算法的推导，证明某些公理的推导类似于谓词公理演算[5]，他们采用的是事例研究的方法来抽取一些在应用程序设计中公共的原则，以应用到以后的类似情形中。他们的方法并不强调预测现存系统的性能，而是指出一种构造新系统的方法；强调的是正确性而不是复杂性。Pepper在文[5]中给出了一种系统的并行算法推导方法以使得并行算法的正确性由构造来保证，该方法采用的技术大部分是原来串行程序开发中使用的，但是作了一些如下扩展：

· 划分数据空间，例如对串结构则将它划分成子串的形式；矩阵结构则划分行或列子矩阵。

· 采用一些高阶算子（如apply-to-all，reduce，zip等）作用在并行进程的通讯网络上。

· 网络上存在反馈回路（操作和额外串的递归所导致）[6]。

在其推导过程中，可以看到这样的一个事实：在对并行算法推导时，对数据空间进行适当划分，针对不同的划分策略推导出具有不同并行性的算法，影射到网络上实现。

五、结 论

Darlington为每个骨架定义了一个性能模型，因此他所定义的骨架被看作为可在一个或多个并行结构中有效执行的高阶函数，我们对Darlington的这种定义作些修改，因为在一个MIMD环境中并行程序设计的主要问题是数据划分问题，因此我们将对这一问题的解答以一种显式的方式加入到骨架中去，即将并行程序设计的数据划分问题和每个骨架结合起来，在选择一个合适的骨架的同时，也意味着选定了一种数据划分的策略，以这一策略来为转换到网络实现提供指导。

在这种情况下，一个骨架除了其自身的代价测度用以指明它的一个有效实现策略外，还表明了对数据的划分方案，以支持其并行执行。以下是已经结合了数据划分的若干骨架：

1. SPEC map_skeleton＝{FUN：★
   AXM VA：S[x]. f★A＝(f★)★split(A)}
2. SPEC zip_skeleton＝{FUN：
   AXM VA，B：S[x]. A B＝split (A) split(B)}
3. SPEC pipe_skeleton＝{FUN：/，
   AXM Vfunction f，g，h：Vx：f(g(h(x)))＝/(f，g，h)}
4. SPEC dc_skeleton＝{FUN dc，indivisible，f
   AXM dc(P)＝IF indivisible(P) THEN f(P)
   ELSE glue(f★split(P))}

其中indivisible：测试程序，是否对象已经是不可分的了；f是求值函数。

可见，采用一种结合了数据划分方案的方法来进行并行算法的推导是一条可行的道路。此外，大多数的简单并行算法都是针对特定的情况来设计的，数据结构只被划分一次，而算法也只作用在一个以这样方式划分的数据上。这种由split和glue来实现的数据划分可看作为静态划分；而在某些情况下（例如更多的复杂算法是由被串行执行的并行成分组成），需要重新进行数据划分，因而动态划分则显得很必要，因此对动态数据划分的研究也是一个比较现实的研究方向。

参考文献

[1]J. Darlington, A. J. Field, P. G. Harrison, Parallel Programming Using Skeleton Functions, Imperial College, London SW7. 2BZ, 1992

[2]Peter Pepper, Mario Sudholt, Jurgen Exner. Functional Programming of Massively Parallel Systems, Technische Universitat, Berlin, July 1992.

[3]Mario Sudholt, Data-Distribution Algebras——A Formal Basis for Programming Using Skeletons, Technische Universitat, Berlin, Dec. 1993

[4]M. I. Cole, Algorithmic Skeletons: Structured Management of Parallel Computation, Pitman/MIT Press 1989

[5]Peter Pepper, Deductive Derivation of Parallel Programs, Technische Universitat, Berlin, July 1992

[6]R. M. Burstall, J. Darlington, A Transfomation System for Developing Recursive Programs, J. ACM, 24, 1977

[7]仲萃豪、冯玉林、陈友君，程序设计方法学，北京科学技术出版社 1985