并行逻辑程序设计语言GHC的实现

王平 胡守仁

(长沙工学院)

IMPLEMENTATION OF THE PARALLEL LOGIC PROGRAMMING LANGUAGE GHC

Wang Ping and Hu Shouren

(*Changsha Institute of Technology*)

ABSTRACT

In this paper, we disigned an abstract parallel inference machine for GHC. We approached and studied a series of subjects about the inference machine, say, process management, environment management, suspending mechanism, committing mechanism, ect., and proposed the corresponding strategy or algorithm. This paper presented a new implementation method for the suspending mechanism. Under the premise of maintaining the efficiency of environment access, the method needs less system overhead and provides the possibility of efficiently implementing GHC.

摘 要

本文设计了一个GHC的并行抽象推理机，对该推理机的进程管理、环境管理、挂起机制、托付机制等一系列课题进行了探讨和研究，并提出了相应的策略和算法。本文提出了一种新的挂起机制实现方法。该方法在保持环境访问效率的前提下，具有较小的系统开销，为GHC的高效实现提供了可能。

§1. 引 言

支持流一与并行的逻辑程序设计语言，如PARLOG、CONCURRENT PROLOG、GUARDED HORN CLAUSE (GHC)等，由于同时面向并发应用和程序的并行执行，在

1989年9月8日收到，1989年11月28日定稿。本文工作得到霍英东教育基金委员会资助。

新一代计算机程序设计语言的研究中具有独特的魅力。日本五代机计划中选定GHC语言为并行推理机核心语言的核心，更为该类语言的发展描绘了一个广阔的前景。因此，对该类语言，特别是GHC 语言，进行研究具有非常重要的意义。

我们在[1] 中对三种典型的并行逻辑程序设计语言—PARLOG、CONCURRENT PROLOG 和GHC 进行了详细的分析。分析结果表明：GHC 较其它两种语言更为成熟，是进一步研究并行逻辑程序设计语言的较好框架。

作为上述工作的继续，本文就GHC 的实现技术进行了深入地研究。本文设计了一个GHC 的并行抽象推理机，对该推理机的进程管理、环境管理、挂起机制，托付机制等一系列课题进行了探讨和研究，并提出了相应的策略和算法。本文提出的新的挂起机制实现方案在保持环境访问效率的同时，还具有较小的系统时空开销，从而为GHC 的高速实现提供了可能。

1.1 GHC 语言简介

GHC 语言是由Ueda 于1985 年提出的一种并行逻辑程序设计语言[2]。其语法非常简单，特点如下：

(1) GHC 程序由带guard 的有限Horn 子句组成。带guard 的Horn 子句形如：

$$p \longleftarrow g_1, \cdots, g_m \mid b_1, \cdots, b_n \quad m, n \geq 0$$

其中P 称为子句头，$g_1, \cdots, g_m$ 称为子句的guard 目标，$b_1, \cdots, b_n$ 为body 目标，“|”称为托付(commitment) 操作符。

(2) GHC 具有两条语义规则：挂起规则和托付规则。违反这两条规则的一致化操作都将被挂起；反之，合乎语义规则的一致化操作都是可并行执行的。

挂起规则：子句的头匹配和guard 目标的执行不能约束调用者中的变量；子句被托付以前，子句体中的body 目标的执行不能约束guard(包括头和guard 目标) 中的变量。

托付规则：给定目标G，当某个子句C 执行该子句的guard 成功，并且确认没有其它子句已经选择给该目标G 时，G 立即托付给子句C。从确认到选定子句这一过程是不可分割的。

1.2 与—或进程模型(详见[6])

1.3 GHC 语言实现中的几个问题

并行逻辑程序设计语言的实现一般涉及到静态编译、环境管理、进程管理、同步机制等问题。我们对此作原则性的分析。

· 静态编译预处理：对GHC 而言，一个语法正确的程序就是合法程序。为优化程序执行，提高执行效率，可增加静态处理。

· 进程管理：并行逻辑程序的执行可用与—或进程模型来描述。基于这种模型的语言实现面临着管理由“与”、“或”两种进程构成的一棵动态“进程树”的复杂任务。

· 环境管理：对于GHC，由于同一与进程(目标G) 的OR 子进程并行执行时不会发生G 中变量被多次赋值的情况，全部一致化操作可在一个环境中进行。因此只存在单一环境。

挂起机制：GHC的挂起机制较复杂。困难的地方在于，当guard目标是嵌套执行的用户定义目标时，如何判断guard目标的执行是否间接约束了目标调用中的变量。

§2. 一个基于与—或进程模型的 GHC 并行推理机—PIM-GHC

图1是GHC推理机的系统结构示意图(该系统结构借鉴了[3]中的研究成果)。

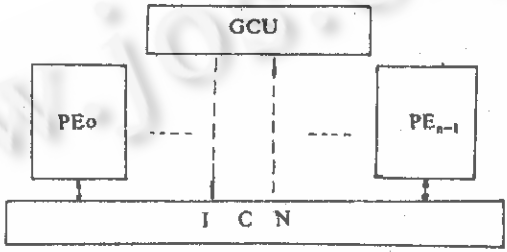

图1　GHC抽象推理机系统结构图

其中，GCU为全局控制部件，负责进程分配、控制命令(进程删除等)的执行等全局控制。ICN为互连网络，为PE之间的通讯提供通道。PE是处理单元，为执行进程的部件。它由进程池、一致化部件、内部谓词部件、静态存贮器和动态存贮器组成。进程池中的进程按挂起进程和就绪进程分为两个队列。

2.1 PE的工作原理

我们可用下述过程描述PE的工作原理

(1) 若PP(进程池)中就绪进程队列(RPQ)为空，则向GCU发送任务请求，等待其它PE传送就绪进程，然后转(2)。

(2) 从RPQ中取出一个进程P送PE。

令P=P(TP, G), TP为进程类型，G为目标。

(3) a. 若G为内部谓词，转(4);

b. 若TP=AND, 则取出程序中对应G的全部候选子句C_i, $i=0,\cdots,k$, $k\geq 0$

若$k=0$, 转(7);

否则对每个子句C_i创建OR进程P(OR(C_i), G), 选留一个OR进程，将其余$k-1$个OR进程送RPQ, P进程挂起，转(3);

c. 若TP=OR(C), C为子句$H \longleftarrow g_1,\cdots,g_m \mid b_1,\cdots,b_n$, $m\geq 0, n\geq 0$。执行H与G的一致化匹配。

若一致化失败，转(7);

若一致化挂起，转(6);

若一致化成功，则

若m=0, 转(8); 否则，为每个g_i创建一个AND 子进程P(AND, g_i), 将P挂起，PE 选留一个AND 子进程，其余$m-1$个子进程送RPQ, 转(3)。

(4) 执行内部谓词，若执行失败，转(7); 若执行挂起，转(6); 若执行成功，转(5);

(5) 执行进程成功处理，转(1);

(6) 执行进程挂起处理，转(1);

(7) 执行进程失败处理，转(1);

(8) 创建P 进程的父进程的父进程的子进程P_i，$i=1,\cdots,n$, 并用来代替P 的父进程。PE 选留一个P_i, 其余$n-1$个送RPQ, 转(3)。

2.2 进程管理

(1) 进程表示

每个进程用一个进程控制块的数据结构来表示，其内容包括:

· 进程名PN

· 进程类型TP, 可为AND, OR(C)

· 目标G, 包括静态文字和环境

· 父进程控制块指针FPCB

· 子进程计数器NSP

· 子句指针C(该指针仅对OR 进程有效)

其中PN 是一个指向一连串子名块的指针。一个子名块包含如下内容:

· 封锁位D, 用于OR 进程托付操作

· 子名SN, 可被进程共享，但不可复制

· 串指钟SP

(2) 进程命名规则

系统为初始进程命令为:

D	SN	SP
0	N_1	nil

如果给定进程P, 其名为PN, 则子进程的名字为 | O | SN | PN |, 其中PN 为进程P 的子句块指针，SN 为系统产生的一个唯一的名字。

(3) 进程调度

对就绪进程进行某种排序，就可实现目标执行的宽度、深度优先，有限深度优先等策略。

(4) 进程删除规则

如果删除名为PN 的或进程，或名为SN —PN 的与进程，则可删除以PN 为名字后缀的全部进程。

(5) 进程创建规则

a. 初始进程由系统创建。给定初始目标G, G= $g_1,\ldots,g_n, n>0$, 系统创建一个OR进程P, 其名为pri, 类型为OR, FPCB=nil, 环境E=nil, 封锁位D=1, 并创建AND 子进程 P_i, $i=1,\ldots,n$, 类型为AND, FPCB=P。其PCB 的关系如图2 所示。

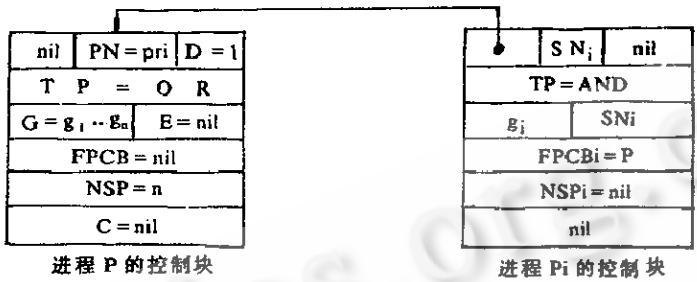

图2　初始进程PCB 的关系图

b. 给定OR 进程P

若D=0, 且进程控制块PCB 如图3(a) 所示。

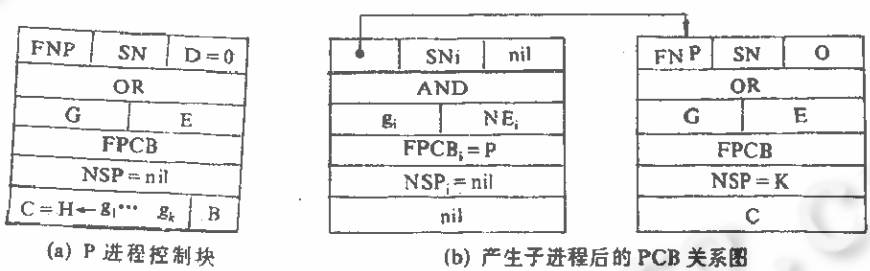

图3　主子进程控制块

P 进程将产生K 个与子进程 P_i, P 进程控制块中的NSP 变为K。子进程控制块中的 NE_i 为H 与P 中目标G 进行一致化时产生的新文本指针。进程控制块间的关系图, 如图3(b) 所示。

若D=1, 且P 的父进程FP≠nil, 设FP 的父进程为FFP, 且子句C=H← $g_1\ldots,g_k|b_1,\ldots,b_n$, $k,n>0$, 则FFP 的控制块中的NSP 值为原值加 $n-1$, 并对应每个 b_i 创建一个子进程 P_i, 如图4 所示。

c. 给定AND 进程P, 设程序中有K 个候选子句: $C_1,\cdots,C_k, K>0$。对应P 中的目标G, 产生K 个或子进程, 并将P 中的NSP 改为K, 如图5 所示。

2.3 环境管理

GHC 环境由多个文本组成。每个文本由两个部分构成: 文本名和文本约束记录表。文本名是系统产生的一个唯一的名字。约束记录表由约束记录构成。每个约束记录形如 $b(v,T,nc,tag)$, 其中 v 是变量符号, T 是项表达式, nc 是文本号, tag 是标志位。

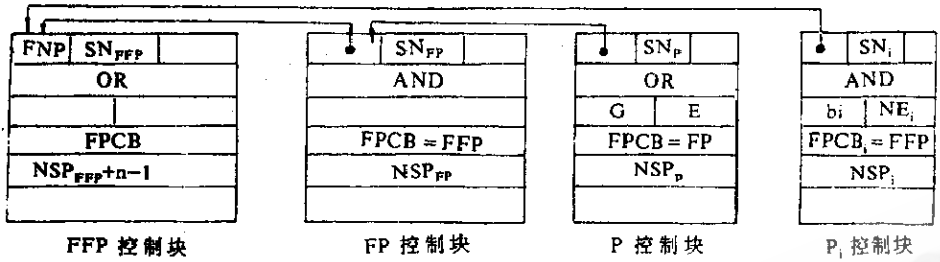

图 4　进程控制块关系图

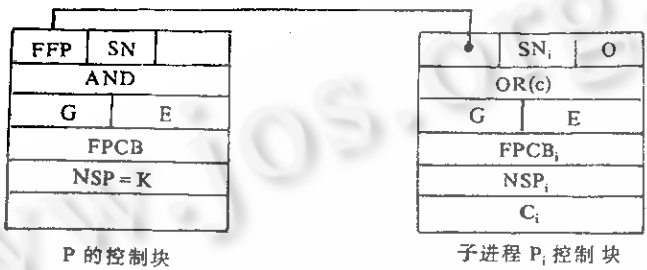

图 5　进程控制块关系图

系统生成初始环境E。E 中只有一个空文本。每执行一次目标和子句头匹配。如果子句头含有变量，则环境中产生一个新文本，子句头中全部变量的约束值都记录在该文本中。子句体中的目标所在的文本号为新文本号。执行内部谓词目标时，不产生新文本，新的变量约束值存放在原内部谓词所在文本中。执行用户目标时，其中的变量符号对应的约束值可以从对应的文本中检索；若查不到约束值，说明该变量未约束。

2.4 挂起机制

在引言中我们已说明，挂起机制的实现难点在于判断嵌套执行中的一致化匹配是否会约束父辈进程目标中的变量。

例如，设有程序：

C1: p(X)⟵q(x)| true.

C2: q(Y)⟵true| Y=a.

⟵p(x).

程序执行时，目标p(X)、q(X) 相继托付给子句C1、C2，当执行body 目标Y=a 时，因对Y 的约束间接约束了p(X) 中的X，故应挂起Y=a。

Ueda 在[2] 中提出了实现挂起机制的设想：当目标中的项与子句头中的变量进行一致化时，将变量指向项的指针打上“染色”标记。仅当头匹配与guard 目标执行完毕，才能将“染色”指针变成“未染色”。一切匹配操作都不能约束利用“染色”指针查询到的项；否则该匹配被挂起。

下面用上述程序的执行环境变化情况说明该设想的工作原理。程序执行步骤与环境对应如下(tag 位为col 时表示已“染色”，为N 时表示“未染色”)。

(1) 目标p(X)与C1的头匹配执行完毕。

(2) guard目标q(X)与C2头匹配完毕。

(3) 执行C2的guard目标，成功。

(4) 执行C2的体目标Y=a，由于查寻Y的值时使用了“染色”指针$X\xrightarrow{col}X, O,$一致化挂起。

环境变化如图6所示。

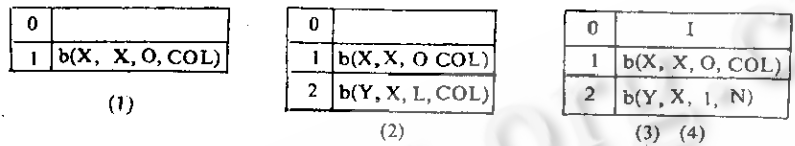

图6　程序执行过程中的环境

但该方法不能正确处理下列程序：

C1: (x) ←—q(x)| true.

C2: q(A(Y))←—true| Y=a.

←—p(A(X)).

按规则，该询问应该被挂起。然而，按上述处理方法，执行完C2的quard目标时环境为：

0	
1	b(X, A(X), O, COL)
2	b(Y, X, O, N)

此时执行Y=a成功，P(A(X))中的变量X被错误赋值。其原因在于，q(x)成功执行完C2的guard后，从Y到X的“染色”指针被错误地“褪色”。对此，我们可做如下改进：

(1) 不允许指针直接超越嵌套的guard目标，而用一串指针链来代替。

(2) 指针可以超越嵌套的guard目标，但必须记住所超越的层数。

(3) 指针可以超越嵌套的guard目标，但它必须记住它最终是为哪个guard目标“染色”的。若该guard成功，则对应该guard目标的“染色”指针都应褪色。

方法(1)简单，但影响环境访问效率；方法(2)要求对应每一染色指针设立整数域，空间和操作开销都较大；方法(3)是较好的方案。经更深入地研究，我们得到一个重要结论：超越嵌套目标的变量约束指针不需“褪色”(详见[4])。

由此，我们给出新的挂起机制实现方案：

(1) 给定目标g，以及子句C: H←—G| B，执行匹配g=H。如果该匹配将g中的变量约束到一个非变量项，或者g中的另一个变量，则一致化操作被挂起；否则，

(2) 对于该匹配产生的变量约束对$V\xrightarrow{p}T$, V、T分别为H中的变量和g中的项。若查找T时使用了“染色”指针，则P为“永久染色”标识；否则，

(3) 将P 打上“染色”标识。

(4) 当guard 执行完毕，guard 所在文本中打上“染色”标识的指针被褪色，即改为“未染色”。

(5) 内部谓词“=”执行时，如产生变量约束对$V \xrightarrow{p} T$, 且查寻v 或T 的过程中都使用了“染色”或“永久染色”指针，则“=”目标挂起；否则，指针P 标识为“未染色”。

(6) 一致化时，经由含有“染色”或“永久染色”指针的指针链查到的项中变量都不能被约束。

我们用下面的程序执行过程说明上述原理。

C1: p(X)←q(X)| true.

C2: q(f(X))←true| x=a.

C3: R(X)←true| x=a.

←P(f(X)), R(X).

执行过程说明如下:

(1) 执行目标P(f(x)) 与C1 的头匹配。

(2) 执行q(x) 与C2 的头匹配。

(3) 执行C2 中目标X=a, 此时被挂起。

(4) 执行R(X) 与C3 的头匹配。

(5) 执行C3 中的目标X=a。

(6) 执行C2 中的目标X=a。

执行时每步的环境变化如图7 所示(图中“C”代表“染色”，“P”代表“永久染色”，“N”代表“未染色”)。

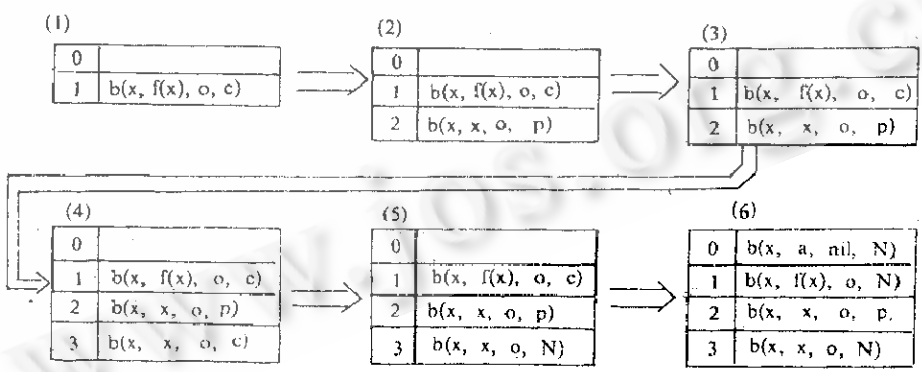

图7　程序执行过程中的环境变化图

2.5 失败、挂起、成功进程的处理

失败进程、成功进程的处理分别如图8、图9 所示。

挂起进程分为两种：一是已产生完所有子进程，正等待子进程回送执行结果(失败或成功) 的进程。对该类进程的挂起处理只需把它的控制块地址存放在其子进程控制块中，保留主控制块即可。另一类是因头匹配一致化被挂起的OR 进程和被挂

起的内部谓词进程。该类进程需挂在进程挂起队列中，在适当的时候，激活这些进程，恢复其运行。

挂起和恢复操作是相互关联的，一般有二种方法：

(1) 忙等待：直接将挂起进程置就绪队列尾，等待下次运行。该方法只需极少的系统开销，但系统运行效率可能较低。

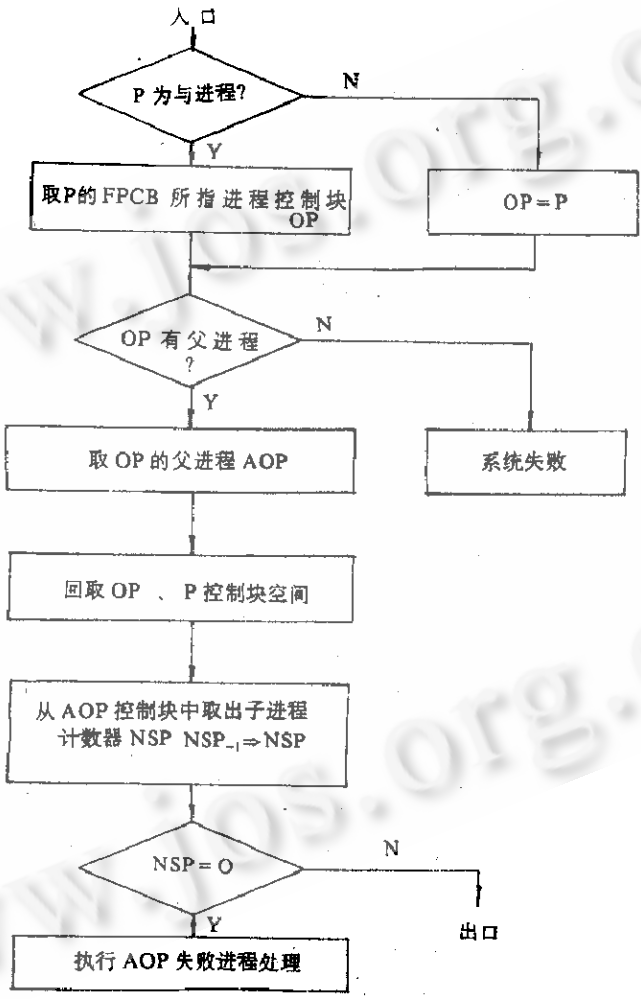

图8　失败进程处理流程图

(2) 进程挂起到引起挂起的变量上。当这些变量被约束，则激活相应的挂起进程。该方案无效计算少，但系统开销较大。

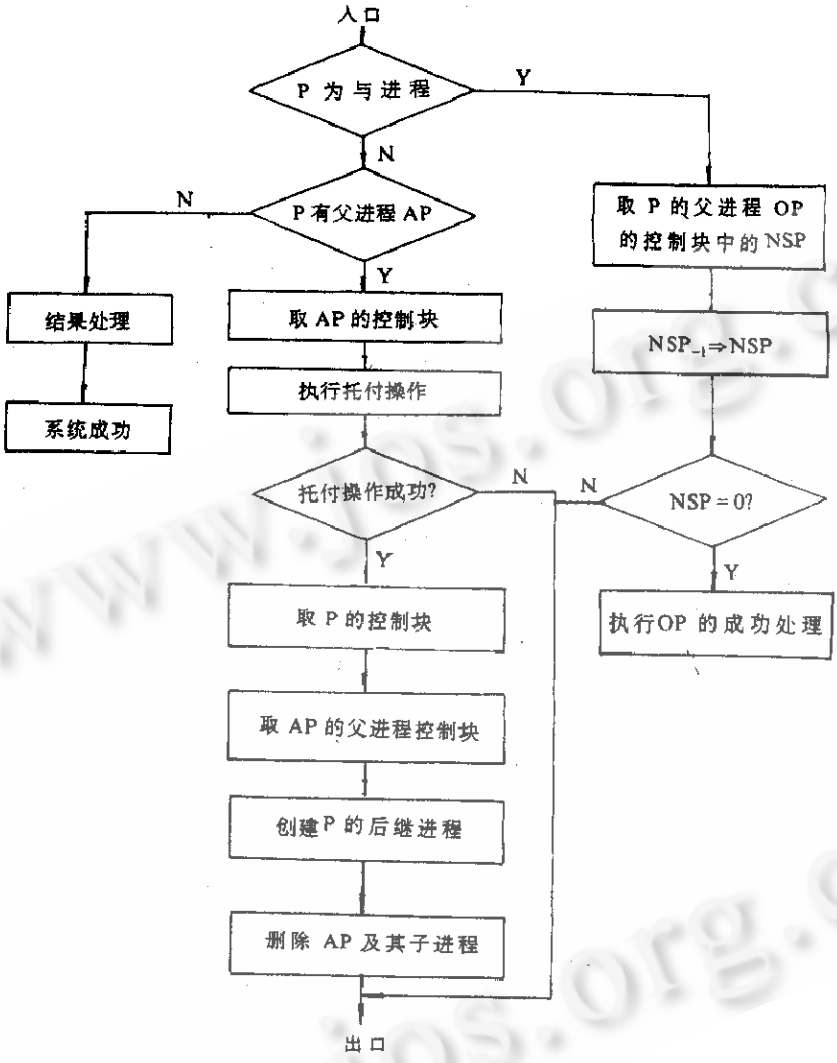

图9　成功进程处理流程图

这是两种极端的方案，如何采取一种折衷方案，使无效计算和系统开销的总和最小，还需做进一步的研究。

2.6 托付操作

托付操作实现较简单，可用图10 的流程描述(其中步骤(3)、(4) 不可分割执行)。

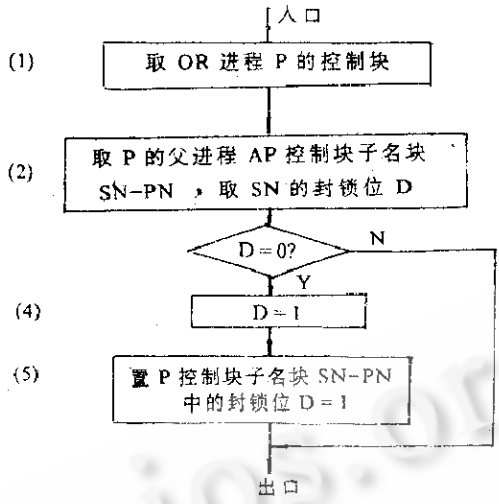

图10 托付操作流程图

§3. 与其它工作的比较

Levy 在1986 年设计了一种GHC 的抽象推理机[5], 同我们设计的PIM-GHC 相比, 它们都是基于与一或进程模型的抽象机, 它们的最大差别在于挂起机制的不同实现技术上。在Levy 的实现中, 约束指针分为两类: “染色”和“未染色”。而我们对“染色”指针又分类为“永久染色”和“染色”两类。这样做的结果, 不仅避免了对“永久染色”指针进行“褪色”, 避免了寻找“染色”指针所属的确切的guard 的工作, 而且还减少了为记录guard 所属“染色”指针所需空间, 进行“褪色”的工作也由于采用文本的环境结构而变得简单: 只需对该文本进行标识“褪色”即可。我们在进行挂起机制操作的全部所需空间只是对每个约束对增加一个三态的标识位, 从而空间开销小。

§4. 结束语

并行推理是新一代计算机的核心技术。作为推理机的核心语言, 必须能够表达通用程序设计中必须的非逻辑成分, 支持程序的高效执行。GHC 尽管是目前较为成熟的并行逻辑程序设计语言, 作为实用语言来要求, 它还必须在上述两方面有重大突破。

GHC 的高效实现技术研究正沿着两个主要方面进行: 一是改进和设计新算法、新策略; 二是在不损害主要功能的前提下简化语言, 以能力丧失为代价换取效率的提高。从实用观点出发, 这种折衷是有效的, 也是实用性研究的一个方向。

参考文献

[1] 王平，胡守仁，“并行逻辑程序设计语言分析”，“863”智能计算机系统89年学术会议，交流论文。

[2] Ueda, K., “Guarded Horn Clauses”, ICOT Tech. Report TR-103, ICOT, 1985.

[3] 孙成政，慈云桂，“PIM-DSOF: A Parallel Inference Machine Based on PSOF Model”, the Second National conf. on Logic Programming, China, 1986.

[4] 王平，“并行逻辑程序设计语言GHC的研究”，国防科大计算机系89届硕士论文。

[5] Jacob Levy, “A GHC Abstract Machine and Instruction Set”, Proc. of the Third Int. Conf. on Logic Programming, 1986.

[6] Akikazu Takeuchi and Koichi Furukawa, “Parallel Logic Programming languages”, Proc. of the Third Int. Conf. on Logic Programming, 1986.

[7] 高耀清，胡守仁，“Design and Implementation of RAP/LOP Parallel Abstract Machine”, Proc. of Int'l Conf. on Info. Processing, Japan, 1990.