维普资讯 http://www.cqvip.com

计算机科学 1996 Vol. 23 No. 2

CSP 语言的再继续语义*)

林 洪
(中国科技大学计算机系 合肥 230027)

王 敏
(中国科技大学电子工程和信息科学系)

摘 要 "Resumption" has been used for specifying parallel programs based on shared memory computational model; but fails to be applied to those based on message passing model, e.g. CSP (Communicating sequential process) language. In this paper, a formalism of constraining the manner of interleaving the execution of commands in concurrent processes is presented, to make resumption qualified to specify CSP programs.

关键词 Denotational semantics, Formal specification, Communicating sequential process, Message passing.

并行程序的指称描述的基本技术是"再继续(Resumption)",用以描述异步并发进程中命令的可能交错[1],这是由一个观察点所能看到的并行程序执行时的行为.用再继续定义基于共享存储(SM)模式的并行程序的指称是合适的.由于信息的一致性由程序保证,命令的任意交错不会产生意外后果.但对基于消息传递模式的并行程序设计语言如 CSP,命令的任意交错则会超出程序的可能行为.在 CSP 程序中,进程中的输入[输出]原语只有和相应进程的输出[输入]原语同时执行,否则就必须等待[2].本文试给出一个约束命令的交错方式的方法,使再继续可以定义 CSP 语言的指称。

1. 抽象文法

我们把注意力集中在 CSP 语言的以下两个主要特征之上:1)一个 CSP 程序是由带标记(Label)的固定数目的顺序进程组成。一个进程不引用或改变另一个与之并行执行的进程中作为赋值对象的变元。2)通过输入(Input)/输出(Output)结构实现通讯和同步。而忽略 CSP 语言的其他一些特征,如分程序结构、卫式命令(Guarded Command)、带矢量下标的进程名、结构赋值命令、输入岗哨(Input Guard)等。

Program ≙ decpart; Declaration_list; body; Command
Declaration_list ≙ Declaration *
Declaration ≙ v:Variable;t:Type
Type ≙ Boolean_type | Integer_type
Command ≙ Skip | Assignment | Compound | Conditional | Loop | Parallel | Input | Output
Assignment ≙ target:Variable;source:Expression
Compound ≙ Command *
Conditional ≙ thenbranch, elsebranch: Command; test:Expression
Loop ≙ body:Command;test:Expression
Parallel ≙ first,second:Process
Input ≙ source:Label;target:Variable
Output ≙ target:Label;value:Expression
Process ≙ l:Label;body:Command
Expression ≙ Constant | Variable | Binary
Constant ≙ Integer_constant | Boolean_constant
Binary ≙ term1,term2:Expression;op:Operator
Operator ≙ Boolean_op | Relational_op | Arithmetic_op
Boolean_op ≙ And | Or
Relational_op ≙ Lt | Le | Eq | Ne | Ge | Gt
Arithmetic_op ≙ Plus | Minus | Times | Div
Boolean_constant ≙ value:B
Integer_constant ≙ value:Z
Variable ≙ id:S
Label ≙ id:S

其中 Parallel、Input、Output 分别定义并行语句、输入命令、输出命令结构.在 CSP 中,Parallel 的一个具体语法是 Label1::block1‖Label2::block2,输入命令和输出命令分别是 Label?variable 和 Label!expression。

这个文法描述的语言是相当简化的,但对本文的目的,已足够适用。

2. 语义定义

2.1 静态语义

对以上定义的 CSP 程序,除了常规的类型检查

*)本文由国家自然科学基金资助。

维普资讯 http://www.cqvip.com

之外，静态语义函数还要完成以下检查：

(1)Parallel 语句中的 Process 是否有相同标记；

(2)Input 和 Output 中的 source 和 target 是否的确标记了在同一个 Parallel 结构中的一个进程；

(3)一个 Parallel 结构中的 Process 是否非法使用了共享变量，即：在一个进程中被赋值的变元在另一个进程中被引用或赋值；

(4)Input 语句中目标变量的类型是否与相应 Output 语句中原表达式的类型一致。

(5)—(7)的静态语义函数不难用常规的方法给出。约束(8)在 CSP 程序中是无法静态定义的，原因如下：

·如果 Input(Output)语句出现在 Conditional 语句中，则可能无法静态确定与之对应的 Output(Input)语句；

·如果 Input(Output)语句出现在 Loop 语句中，则可能无法静态确定其执行次数。

作为 CSP 家族的一员，Occam 不标识进程，但在分程序首部定义通道(Channel)名和类型，Input 和 Output 语句引用通道名，于是(8)可以静态检查。

以下转入动态语义的讨论。

2.2 动态语义

对本文定义的 CSP 语言，除了要给出其中常规结构的含义函数之外，动态语义规范还要定义 Parallel 的指称，如果采用再继续，则要限制命令的交错方式，以模型化 Input 和 Output 语句的行为。

2.2.1 动态语义论域

状态(State)除了模型变量的当前值之外，还要标记是否是一个输入/输出状态，这导致引入子论域 Sending_state 和 Receiving_state。此外，在 State′中还包含一个死锁状态 Deadlock，它表示输入/输出语句无法执行的状态。

```
Value ≜ Z ∪ B ∪ {unknown}
Sending_state ≜ l:Label⊥;v:Value
Receiving_state ≜ l:label⊥;v:Variable
State ≜ vm:Variable→Value;sending:Sending_
  state;receiving:Receiving_state
Resumption ≜ State→P(State′)
State_and_Resumption ≜ state:State;resumption:
  Resumption
State′ ≜ State | State_and_Resumption | Deadlock
```

其中 Label⊥ Label ∪{⊥}，Sending_state 中的 v 域记录要输出的表达式值，l 域记录目标进程的标记，若 l 为⊥，则表示非输出状态；Receiving_State 中的 v 域记录需被赋值的变量，l 域记录源进程的标记，若 L 为⊥，则表示非输入状态。

Resumption 模型化一组可能的计算序列，State_and_Resumption 记录当前状态和未完成的计算步序列。

2.2.2 动态语义函数

(1)一些辅助函数。Step 用来建立单步的计算序列；Pause 定义允许在完成任何动作前被打断的序列，它的作用是显式地建立断点；两个再继续的计算步的复合由"*"定义：

```
step:(State→State)→Resumption
step(f)≜ λ σ.{State′(f(σ))}
pause:Resumption→Resumption
pause(r)≜ λ σ.{State′(State_and_Resumption
  (state:σ;resumption:r))}
infix"*":Resumption×Resumption→Resumption
r1*r2≜ r1;(λ σ′:State′.
  case σ′ of
    State→{State′(State_and_Resumption(state:
      σ′;resumption:r2))}|
    State_and_Resumption→{State′(State_and_
      resumption
      (state:σ′.state;resumption:σ′.resumption
      *r2))}|
    Deadlock→{State′(Deadlock)}
  end)⊥
```

Parallel 定义形如 l1::P1‖l2::P2的语句的含义。如果进程的 p1,p2的语义分别是 r1,r2，parallel(r1,r2,l1,l2)给出 r1,r2的所有可能的交错，但 p1(p2)中一旦有形如(l2(l1)?variable 或 l2(l1)!expression 的语句，[即对初始状态 σ,r1(σ)(或 r2(σ))中包含这样的 State′，它是输入输出状态 σ′或以 σ′为当前状态的 State_and_Resumption(state:σ′;resumption:r)]，则它们的执行必须和 p2(p1)中的形如 l1(l2)!expression 或 l1(l2)?variable 的语句的执行同步，其他任何交错方式都将导致 Deadlock(即此时 r2(或 r1)必须作用于 σ′，且 r2(σ′))(或 r1(σ′))中只包含这样的 State′，它为 Deadlock，或者不以输入/输出状态为当前状态)。输入/输出状态的形成由 Input 和 Output 的含义函数完成。

```
parallel:Resumption×Resumption×Label×Label
  →Resumption
parallel(r1,r2,l1,l2)≜
 given
  then ≜ λ r1,r2,l2,l1:(λ σ′:State′.
   case σ′ of
    State→
      if σ′.sending.l=l2 then(λ σ:State′.
       case σ of
         State→if σ.sending.l=⊥then{State′(σ)}
           else{State′(Deadlock)}
           end|
          State_and_Resumption→
           if σ.state.sending.l=⊥ then {State′
             (σ)}
           else{State′(Deadlock)}
           end|
          Deadlock→{State′(Deadlock)}
        end)+r2(σ′)
      elsif σ.receiving.l=l2 then(λ σ:State.
       case σ of
        State→if σ.receiving.l=⊥ then{State′
```

维普资讯 http://www.cqvip.com

```
            (σ)}
           else{State'(Deadlock)}
           end|
          State_and_Resumption→
           if σ. state. receiving. l = ⊥ then {State'
             (σ)}
           else{State'(Deadlock)}
           end |
          Deadlock→{State'(Deadlock)}
         end)⊥r2(σ')
        else {State'(State_and_Resumption (state:
         σ';resumption:r2))}
        end |
       State_and_Resumption→
      if σ'. state. sending. l=l2 then(λ σ;state.
       case σ of
        State→
         if σ. senidng. l = ⊥ then
          {State'(State_and_Resumption
          (state:σ;resumption:σ'. resumption))}
         else{State'(Deadlock)}
         end |
        State_and_Resumption→
         if σ. state. sending. l = ⊥ then
          {State'(State_and_Resumption (state:σ.
            state;resumption;then(σ'. resumption,
            σ. resumption)))}
          {State'(State_and_Resumption (state:σ.
            state;resumption;then(σ. resumption,
            σ'. resumption)))}
         else{State'(Deadlock)}
         end |
        Deadlock→{State'(Deadlock)}
       end)⊥r2(σ')
      elsif σ'. state. receiving. l=l2 then(λ σ;State'.
       case σ of
        State→
         if σ. receiving. l = ⊥ then
          {State'(State_and_Resumption
           (state:σ;resumption:σ'; resumption))}
         else{State'(Deadlock)}
         end |
        State_and_Resumption→
         if σ. state. receiving. l = ⊥ then
          {State'(State_and_Resumption (state:σ.
           state;resumption;then(σ'. resumption,σ.
           resumption)))}
          {State'(State_and_Resumption (state:σ.
           state;resumption;then(σ. resumption,σ'.
           resumption)))}
         else{State'(Deadlock)}
         end |
        Deadlock→{State'(Deadlock)}
       end)⊥ r2(σ')
      else
       {State'(State_ and_ Resumption (state: σ'.
        state; resumption; then (σ'. resumption,
        r2)))}
       {State'(State_ and_ Resumption (state: σ'.
        state; resumption; then ( r2. σ'.
        resumption)))}
      end |
     Deadlock→{State'(Deadlock)
    end)⊥
then
  λ σ. (then(r1,r2,l2)(σ))∪(then(r2,r1,l2)(σ))
end
```

可以看到 parallel 定义的合理性由以下状态不变式保证，同时它也保此状态不变式：

$$\forall\ \sigma:\mathrm{State}.\ (\sigma.\mathrm{sending}.l = \bot \vee \sigma.\mathrm{receiving}.l = \bot) \tag{1}$$

函数 flatten 完成从一个再继续到一组不可打断的计算序列的变换：

```
flatten:Resumption→State→P(State)
flatten(r)≜r;(λ σ';State'.
  case σ' of
   State→{σ'}|
   State_and_Resumption→(flatten(σ;', resump-
     tion))(σ'. state)|
   Deadlock→φ
end)⊥
```

如果再继续 r 作用于状态 σ 后得到的 State' 集中有 Deadlock 状态，则相应的计算序列得不到任何出口状态，这就是 flatten 定义中 Deadlock→φ 分支的含义。

(2)含义函数。在这个语言中，命令的指称是一个再继续，因为命令可能嵌在一个并行计算中，一个命令的计算序列可能已被这个并行构造的其他部分错开。如果命令构成程序，它的再继续被打断为一族可能的计算序列。

```
M_Command:Command→Resumption
M_Command[C]≜
 case C of
  Skip→M_Skip[C]|
  Assignment→M_Assignment[C]|
  Compound→M_Compound[C]|
  Conditional→M_Conditional[C]|
  Loop→M_Loop[C]|
  Parallel→M_Parallel[C]|
  Input→M_Input[C]|
  Output→M_Output[C]
end
```

下面是各种命令的定义函数：

Skip 的含义是，维持当前状态不变：

```
M_Skip:Skip→Resumption
M_Skip[s]≜step(λ σ. σ)
```

Assignment 的含义是：改变当前状态中被赋值变量的值，维持其他成分不变：

```
M_Assignment1:Assignment→Resumption
M_Assignment2[a]≜
   step (λ σ. (σ except (vm: σ. vm ∪ {a. target,
     M_Expression[a. source](σ)})))
```

其中 $M_{Expression}$ 计算一个表达式在当前状态下的值，是原子操作：

```
M_Expression:Expression→State→Value
M_Expression[e]≜λ σ:State.
  case e of
   constant→
    case e of
     Integer_constant→e. value|
     Boolean_constant→e. value
    end|
   Variable→
```

维普资讯 http://www.cqvip.com

```
    if e ∈ dom σ. vm then unknown
    else(σ. vm)(e)end
  Binary→
    apply_binary(e. op.
      M_Expression[e. term1](σ),
      M_Expression[e. term2](σ))
end
```

于是赋值过程不可被打断。

因为条件命令可以在测试后且子句计算前被打断，所以 Pause 函数必须插入以建立显式断点，循环命令的处理类似：

```
M_Conditional : Conditional→Resumption
M_Conditional[c] ≜ λ σ:State.
  if M_Expression(c. test)(σ) then pause(M_Command[c.
    thenbranch])(σ)
  else pause(M_Command[c. elsebranch])(σ)
  end
M_Loop : Loop→Resumption
M_Loop[l] ≜ λ σ:State.
  if M_Expression[l. test](σ) then pause(M_Command[l.
    body]*M_Loop[l](σ)
  else step(λ σ. σ)(σ)
  end
```

Compound，Parallel 语句的含义是：

```
M_Compound : Compound→Resumption
M_Compound[cp] ≜ over cp apply M_Command combine"*"
  empty step(λ σ. σ)end
M_Parallel : parallel→Resumption
M_Parallel[p] ≜ parallel(M_Process[p. first], M_process[p.
  second], p. first. l, p. second. l)
M_Process : Process→Resumption
M_Process[p] ≜ M_Command[p. body]
```

Input 和 Output 的含义是：如果当前状态不是输入或输出状态，则把它变成输入或输出状态；如果是输入或输出状态，且该状态"要求"该语句的执行，则"执行"该语句，并把状态变成非输入输出状态，否则维持该状态不变，并在该状态之后附加一个计算步。这样便维持了通讯的"顺序"性：

```
M_Input : Input→Resumption
M_Input[i] ≜ λ σ:State.
  if σ. sending. l = ⊥ and σ. receiving. l = ⊥ then
    {State'(State (σ except (receiving:Receiving
    state(l;i. source;v;i. target))))}
  elsif σ. sending. l = i. source then {State'(State
    (σ except (vm:σ. vm ⋃ {i. target, σ. sending. v};
    sending:σ. sending except(l;⊥))))}
  else {State'(State_and_Resumption(state:σ;re-
    sumption:M_Input[i]))}
  end

M_Output : Output→Resumption
M_Output[o] ≜ λ σ:State.
  if σ. sending. l = ⊥ and σ. receiving. l = ⊥ then
    {State;'(State (σ except (sending:Sending_
    state(l;o. target;v;o. value)))}
  elsif σ. receiving. l = o. target then,State'(State(σ
    except(vm:σ. vm {σ. receiving. v, o. value\;re-
    ceiving:σ. receiving except(l;⊥))))}
  else {State'(State_and_Resumption(state:σ;re-
    sumption:M_Output[o]))}
end
```

Input 和 Output 保状态不变式(1)。如果 Input 和 Output 出现在一个 Parallel 结构中，如 P=*l*1::p1‖*l*2::p2，且 p1=…*l*2?x;…，p2…*l*1!e;…则 P 的执行在到达 *l*2?x 语句时必须转入 p2，直至执行到 *l*1!e 以消除输入状态为止，在此期间交错不可发生，否则将导致 Deadlock，如果输入状态不能消除(如在 *l*1!e 之前遇上其他输入输出语句且该语句不能执行)，也导致 Deadlock。若先执行 *l*1!e，情况类似。上述过程示于图1。

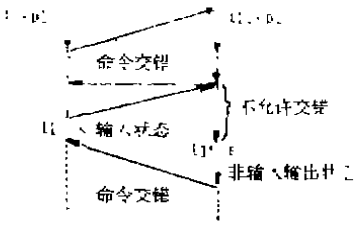

图1 输入输出语句对交错的限制

最后，Program 的含义是：

```
M_Program : Program→State→P(State)
M_Program[p] ≜ flatten(M_Command[p. body])
```

可以证明，对于满足静态约束的任意 CSP 程序 P 和任意状态 σ，$M_{program}[p](\sigma)$的值为 P 或某个确定的非输入输出状态，前者表示程序 P 从状态开始的运行将进入死锁。

3. 结论

本文定义了一个简单的 CSP 语言的指称语义规范，通过约束再继续的交错方式描述了含输入/输出语句的进程的含义。

不难把本文的工作推广到完整的 CSP 语言上，由此也不难定义具体语言如 Occam 的指称规范。如果再引进"存储(Store)"的概念，还可以定义兼容同步和异步通讯模式的并行语言的语义[3]。

致谢 陈意云老师审阅了本文，提出了宝贵的修改意见，并指出了深入研究的必要和方向。在此深表谢意！

参 考 文 献

[1]陈意云，形式语义学基础，中国科学技术大学出版社，1994

[2] Hoare, C. A. R., Communicating Sequential Processes. in: S. L. Graham, R. L. Rivest eds., Communications of the ACM, 21(8), 1978

[3] Mosses, P. D., Denotational Semantics. in: J. van Leeuwen, ed., Handbook of Theoretical Computer Science, Vol. B (North-Holland, Amsterdam, 1990)