

variable sharing と message sending との間の プログラム変換

都立大 理 足田輝雄

東大 理 石畑清

1. はじめに

高級言語による並列処理プログラミングの必要性が増大するにつれて、プロセス間の通信と同期のための各種の言語プリミティブが提案されてきている。たとえば、セマフォア, path expression, モニター, communicating sequential process, distributed process, Ada におけるランデブーなどがある。これらのメカニズムそれぞれが、通常の並列プログラムを表現するのに充分な能力をもっている。つまり、あるメカニズムは別のメカニズムによってインプリメントできる。

これらのメカニズムにおける 2 つの基本的なカテゴリーといえるべきものが, variable sharing と message sending である。これら両者間の関係を調べたいというのが本稿の大きな目標であるが、ここではとくに、両者のプログラムの間の変換の方法を提案する。具体的には、「モニター変換」と名づけ

る変換スキームと, それらを用いた変換戦術とを提案する.
 この方法は言語ポリミティクのインフォーマントではなく, む
 しろソース・プログラム段階の変換である. 本稿ではこの方
 法を適用することによって, 'dynamic sorting array' の2つの
 プログラムの同値性を示すことにする.

2. モータ - 変換

2.1. ポリミティク

ここでもまず, 本稿で用いるポリミティクを決めておく. わ
 れわれは両メカニズムの基本的な点についてのみ関心がある
 ので, ここでのポリミティクは比較的簡単な機能をもつもの
 とする.

まず variable sharing のポリミティクとしては次の2つの文
 とする:

await B

await B then $V \leftarrow E$

ここで B は論理式, V は変数, E は式である. これらの
 文の意味は自明である. たとえば Dijkstra の $P(s)$ や
 $V(s)$ は次のようにかける:

$P(s) :$ await $s = 1$ then $s \leftarrow 0$

$V(s) :$ $s \leftarrow 1$

message sending のフォーマティヴはほとんど Hoare の CSP の考えに従う。すなわち次の2つの文を用いる:

send(val) to process1

receive(var) from process2

通常の message sending の方式の提案によく見られるように、次のような「高級」な仕様は扱わない:

- ・メッセージの自動的なバッファリング
- ・パターンマッチによるメッセージの引渡し (シグナルだけは例外)
- ・エントリ (Ada) やポートのような概念

メッセージを受取る際には、言語仕様として、ノンデターミニズムを許すことにする。記法としては Dijkstra のゲート・コマンドを用いる。

2. 2. モーター・スキーム

Scheme 1 において、上半部の、message sending による並列プログラムは、下半部の variable sharing によるプログラムと同値であると思える。この変換によって前者の send 文と receive 文とを消去することができる。

ここで注意することは、前者のプログラムにおけるプロセス M は、Brinch Hansen と Hoare が提案したモーターである

```

process M =
begin
  do
    receive(req1()) from P1 → S1; send(val1) to P1
  or
    receive(req2()) from P2 → S2; send(val2) to P2
  od
end;

```

```

process P1 =
begin
  ...
  send(req1()) to M; receive(var1) from M;
  ...
end;

```

```

process P2 =
begin
  ...
  send(req2()) to M; receive(var2) from M;
  ...
end

```



```

shared s;

```

```

process P1 =
begin
  ...
  await s = 1 then s ← 0;
  S1; var1 ← val1;
  s ← 1;
  ...
end;

```

```

process P2 =
begin
  ...
  await s = 1 then s ← 0;
  S2; var2 ← val2;
  s ← 1;
  ...
end

```

Scheme 1. With input guards.

と見なされうることである。すなわち、プロセス M は、いくつかの変数と、メッセージを受取った際にそれらの変数に「相互排除」的に作用するいくつかの手続き（モーター手続き）とからだけで成立っていて、プロセス自身の 'private code' をもっていない。

このスキームは、実際、モーターのセマフォアによるインフォーマントであるとも見なされ、それはこゝまでよく知られているものである。たとえば Ada のランデブーに対しては、accept 文のセマフォアによる（もっと複雑な）インフォーマントが Habermann と Nassi [4] によって与えられている。

このように上の同値はインフォーマントとしては新しいはなすが、われわれの視点は、これをむしろ基本的な変換スキームとして採用しようということである。互いに対応する send 文と receive 文とがごく 自然 にキャンセルできる場合があり、それがモーター・スキームであるというわけである。

Scheme 1 は少し一般化されうる。それが Scheme 2 である。これは後での例の中で実際に用いる。また逆にある場合にはこれらのスキームはいくらか単純化されうる（ここではふれない）。

いわゆるモーター・プロセスに対してはつねにこれらのスキームが互いに適用できるので、プログラムの変換例はいく

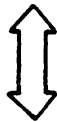
```

process M =
begin
  do
    B1; receive(req1()) from P1  $\rightarrow$  S1; send(val1) to P1
  or
    B2; receive(req2()) from P2  $\rightarrow$  S2; send(val2) to P2
  od
end;

process P1 =
begin
  ...
  send(req1()) to M; receive(var1) from M;
  ...
end;

process P2 =
begin
  ...
  send(req2()) to M; receive(var2) from M;
  ...
end

```



```

shared s;

process P1 =
begin
  ...
  await B1 and s = 1 then s  $\leftarrow$  0;
  S1; var1  $\leftarrow$  val1;
  s  $\leftarrow$  1;
  ...
end;

process P2 =
begin
  ...
  await B2 and s = 1 then s  $\leftarrow$  0;
  S2; var2  $\leftarrow$  val2;
  s  $\leftarrow$  1;
  ...
end

```

Scheme 2. With Boolean and input guards.

も作れる。しかしこの変換(同値)は新しいものではな
ので、ここでは具体例は省く。(モーター・プロセスの例は
たとえば [9] を参照。)

3. モーター変換の適用

3.1. 一般的な戦術

ここで行われるのは、message sending を用いた並列プログラ
ムから variable sharing を用いたプログラムへの変換のため
の戦術を提案する。これは次のように述べられる。

Phase 1. あるプロセスをより小さなプロセスに「分割」
し、各小プロセスがスキームにおけるプロセス
SM と同じ形をもつように変形する。

Phase 2. スキームを用いて send 文と receive 文とをキ
ャンセルし、その小プロセスを消去する。

実際には phase 1 の段階が必ずしも、一般に工夫を要し、
また成功するとはかぎらない。そのあたりの状況は次に述べ
る例で見よう。

3.2. Dynamic sorting array

ここでは、'dynamic sorting array' とよばれるアルゴリズムの、
2つの版の同値をプログラム変換によって示すことにする。

access-to-shared

```

here[i], here[i+1] : array [1..2] of int; length[i], length[i+1] : 0..2;
{ initially, length[i] = length[i+1] = 0 }

```

process sort[i] =

```

var rest : int;

```

begin

```

rest ← 0;

```

do

```

await length[i] = 2 or (length[i] = 0 and rest > 0);

```

```

if length[i] = 2 then

```

```

    if here[i][1] > here[i][2] then swap(here[i][1], here[i][2]) fi;

```

```

    if rest > 0 then await length[i+1] = 1 fi;

```

```

    here[i+1][length[i+1]+1] ← here[i][2];

```

```

    length[i+1] ← length[i+1] + 1;

```

```

    length[i] ← 1; rest ← rest + 1

```

```

else { length[i] = 0 and rest > 0 }

```

```

    await length[i+1] = 1;

```

```

    here[i][1] ← here[i+1][1];

```

```

    length[i+1] ← 0;

```

```

    length[i] ← 1; rest ← rest - 1

```

fiodend

Program B. Dynamic sorting array — variable sharing.

```

process sort[i] =
  var here : array [1..2] of int; length : 0..2; rest : int;
begin
  length ← 0; rest ← 0;
  do
    receive(here[length+1]) from sort[i-1] →
      length ← length + 1;
      if length = 2 then
        if here[1] > here[2] then swap(here[1], here[2]) fi;
        send(here[2]) to sort[i+1];
        length ← 1; rest ← rest + 1
      fi
    or
    receive(get()) from sort[i-1] →
      send(here[1]) to sort[i-1];
      length ← 0;
      if rest > 0 then
        send(get()) to sort[i+1]; receive(here[1]) from sort[i+1];
        length ← 1; rest ← rest - 1
      fi
    od
  end

```

Program A. Dynamic sorting array — message sending.

このアルゴリズムは Brinch Hansen によるもので、かれの distributed process の例として用いられたものである [1]。プログラムはプロセス $\text{sort}[i]$ の 1 次元的な配列からなっており、各プロセスは通常の状態であらう 1 つの値を保持することになる。Program A. がかれかれの message sending 版であり、Program B. が variable sharing 版である。(もちろんこれはもとのものとは異なる。)

かれかれは Program A から出発する。各プロセスの (セマンティカルな) 構造を見ることにより、ノンデターミニズムをもっととりこめて、Program A は Step 1 のように変形できる。(この変換は実際、非常に「セマンティカル」である)

次にかれかれは、プロセス $\text{sort}[i]$ を 2 つの subprocess $\text{sortreceiver}[i]$ と $\text{sortsender}[i]$ とに分割する。これを行ううために新たにセマフォ変数 $t[i]$ を導入し、これら 2 つのプロセス間で共有されることになる変数 $\text{here}[i]$, $\text{length}[i]$, $\text{rest}[i]$ に対する「相互排除」を実現する。これが Step 2 である。

次に、プロセス $\text{sortreceiver}[i]$ において、これをモーター・プロセスの形に変形する。このためには変数 $t[i]$ を「くり返」めばよい。そして Step 3 を得る。

次はたんに Scheme 2 を適用するだけである。かれかれは

```

process sort[i] =
  var here : array [1..2] of int; length : 0..2; rest : int;
begin
  length ← 0; rest ← 0;
  do
    (length = 0 and rest = 0) or length = 1;
    receive(here[length+1]) from sort[i-1] →
      length ← length + 1
  or
    length = 2 →
      if here[1] > here[2] then swap(here[1], here[2]) fi;
      send(here[2]) to sort[i+1];
      length ← 1; rest ← rest + 1
  or
    length = 1; receive(get()) from sort[i-1] →
      send(here[1]) to sort[i-1];
      length ← 0
  or
    length = 0 and rest > 0 →
      send(get()) to sort[i+1]; receive(here[1]) from sort[i+1];
      length ← 1; rest ← rest - 1
  od
end

```

Step 1.

```

shared  { by sortreceiver[i] and sortsender[i] }
    here : array [1..2] of int; length : 0..2; rest : int; t : sema;
    { initially, length = 0, rest = 0, and t = 1 }

process sortreceiver[i] =
begin
    do
        await ((length = 0 and rest = 0) or length = 1) and t = 1 then t ← 0;
        if
            receive(here[length+1]) from sortsender[i-1] →
                length ← length + 1
            or
                receive(get()) from sortsender[i-1] →
                    send(here[1]) to sortsender[i-1];
                    length ← 0
            fi;
        t ← 1
    od
end;

process sortsender[i] =
begin
    do
        await (length = 2 or (length = 0 and rest > 0)) and t = 1 then t ← 0;
        if length = 2 then
            if here[1] > here[2] then swap(here[1], here[2]) fi;
            send(here[2]) to sortreceiver[i+1];
            length ← 1; rest ← rest + 1
        else { length = 0 and rest > 0 }
            send(get()) to sortreceiver[i+1]; receive(here[1]) from sortreceiver[i+1];
            length ← 1; rest ← rest - 1
        fi;
        t ← 1
    od
end

```

Step 2.

```

process sortreceiver[i] =
begin
    do
        ((length = 0 and rest = 0) or length = 1) and t = 1;
        receive(here[length+1]) from sortsender[i-1] →
            t ← 0;
            length ← length + 1;
            t ← 1
    or
        length = 1 and t = 1;
        receive(get()) from sortsender[i-1] →
            t ← 0;
            send(here[1]) to sortsender[i-1];
            length ← 0;
            t ← 1
    od
end

```

Step 3.

access-to-shared

```

here[i], here[i+1] : array [1..2] of int; length[i], length[i+1] : 0..2;
rest[i], rest[i+1] : int; t[i], t[i+1] : sema;
{ initially, length[i] = length[i+1] = 0, rest[i] = rest[i+1] = 0,
  and t[i] = t[i+1] = 1 }

```

process sort[i] =

begin

do

await (length[i] = 2 or (length[i] = 0 and rest[i] > 0)) and t[i] = 1 then t[i] ← 0;

if length[i] = 2 then

if here[i][1] > here[i][2] then swap(here[i][1], here[i][2]) fi;

await ((length[i+1] = 0 and rest[i+1] = 0) or length[i+1] = 1) and t[i+1] = 1;

here[i+1][length[i+1]+1] ← here[i][2];

t[i+1] ← 0; length[i+1] ← length[i+1] + 1; t[i+1] ← 1;

length[i] ← 1; rest[i] ← rest[i] + 1

else { length[i] = 0 and rest[i] > 0 }

await length[i+1] = 1 and t[i+1] = 1;

t[i+1] ← 0; here[i][1] ← here[i+1][1]; length[i+1] ← 0; t[i+1] ← 1;

length[i] ← 1; rest[i] ← rest[i] - 1

fi;

t[i] ← 1

od

end

Step 4.

直ちに Step 4 を得る。

最後に、各 $\text{sort}[i]$ において、同期のためには、セマフォ $t[i]$ でなくとも変数 $\text{length}[i]$ の値を見張っていかねばできることに気がつけば（これはセマンティカルな発見であるが）、 $t[i]$ は無用になり、目標の Program B が得られる。

この変換全体において、もっとも重要な部分は、Program A から Step 1, Step 2 あたりまでの、プロセスを分割していくところである。

4. おわりに

並列プロセス間における critical section の実現の問題

(Dijkstra) の解はたくさんあるが、その中で Lamport による有名な bakery algorithm (variable sharing) [6] と、さきさんの Ricart-Agrawala によるもの (message sending) [7] とは、アイデアが類似しており、実際それぞれの方法で両者の（ほとんど）同値であることが示せる。（変換過程がいささか長くなるのでここでは示すことはやめた。）

本稿の方法は非常に ad hoc なし informal であり、つまり mechanical でない。この点は今後の改善点である。

ともあれ、本稿の方法自身が、variable sharing と message sending との関係、あるいはその中でのモーターの位置を、何

3つの意味で示唆してゐるようにならる。

REFERENCES

1. P. Brinch Hansen, 'Distributed processes: a concurrent programming concept', Comm. ACM, 21, 934-941 (1978).
2. M. Broy, 'Transformational semantics for concurrent programs', Inform. Process. Lett., 11, 87-91 (1980).
3. W. Eventoff, D. Harvey and R. J. Price, 'The rendezvous and monitor concepts: is there an efficiency difference?', Proc. of the ACM-SIGPLAN Symp. on the Ada Programming Language, SIGPLAN Notices, 15 (11), 156-165 (Nov. 1980).
4. A. N. Habermann and I. R. Nassi, 'Efficient implementation of Ada tasks', Dept. of Computer Science, Carnegie-Mellon Univ., Pittsburgh, Pa. (Jan. 1980).
5. C. A. R. Hoare, 'Communicating sequential processes', Comm. ACM, 21, 666-677 (1978).
6. L. Lamport, 'A new solution of Dijkstra's concurrent programming problem', Comm. ACM, 17, 453-455 (1974).
7. G. Ricart and A. K. Agrawala, 'An optimal algorithm for mutual exclusion in computer networks', Comm. ACM, 24, 9-17 (1981).
8. United States Department of Defense, Reference Manual for the Ada Programming Language, Proposed Standard Document, July 1980.
9. J. Welsh and A. Lister, 'A comparative study of task communication in Ada', Software - Practice and Experience, 11, 257-290 (1981).