# 第4章　副プログラム境界でのデータマッピング

この章では、「呼出し元は...しなければならない」という言い回しは、実装 (すなわち、コンパイラにより生成されるコード) に対する制約を表すもので、プログラマの記述するソースコードに対するものでない。

## 4.1　はじめに

この節では、マッピング指示文が、副プログラムへ受け渡される引数とどのように関連するのかについての概要を示す。ただし、ここで用いられる言葉は、完全なものではない。厳密な規則はこの節に続くいくつかの節において説明する。

第 3 章で述べたデータマッピング機能に加えて、HPF では仮引数のマッピングに関して多くのオプションが用意されている。

各々の仮引数のマッピングは、それを呼び出す主プログラム又は手続 (“呼出し元”) での対応する実引数と、いくつかの異なった方法で関連づけられる。このために、仮引数に対するマッピング指示文は、3 通りの異なる構文的な形をとることができる。すなわちそれは、**指令的**、**記述的**、**転写的**の 3 つである。

HPF の提供するこれら 3 つの記法は、プログラマが、データをそのままの場所に置いておくのか、あるいは呼出しの間だけ、新しい、そしておそらくは呼ばれた側の副プログラムの実行がより効率よく行なえるようなマッピングに、データを自動的に再マップするのかを指定できるようにする。

これらの記法の意味は次の通りである。

**指令的** 指令的な指示文は、仮引数のマッピングについて指定する。ただし、実引数はそれと同じマッピングを持たなくても構わない。同じマッピングを**持たない**場合、指定された通りに引数を再マップするコードを生成し、副プログラムの出口で元のマッピングに戻すコードを生成することは、コンパイラの責任となる。このコードは、呼出し元に生成される場合もあれば、呼ばれた側の副プログラム中に生成されることもあるだろう。第 4.6 節で説明する明示的引用仕様の必要性により、マッピングをどちらの側で行なうにしても、必要な情報はコンパイル時に得ることが可能である。

指令的な指示文は、構文的にはプログラムの他の部分に現われる指示文と同じである。例えば、A が仮引数だとすると、

```
!HPF$ DISTRIBUTE A (BLOCK, CYCLIC)
```

は、指令的な指示文となる。

**記述的** 記述的な指示文は、指令的な構文と全く同じ意味を持つ。違うのは、記述的な指示文は、実引数が再マッピングを必要としないというプログラマによる弱い表明になるということである。

この表明が、“弱い” と表されるのは、もしその表明が偽であっても、プログラムは依然として標準規格合致であるからである。そのような場合、コンパイラは、適切な再マッピングを生成しなければならない。

もしコンパイラが、その表明が偽だと判定できた時、あるいはその表明が真であるかどうか確定できない時は、警告か、診断情報メッセージを出力できる。

> 【利用者への助言】 指令的な指示文に対して記述的な指示文があることの目的は、プログラムの開発及びデバッグのために有用な情報を、コンパイラがプログラマに報告する手段を、単に提供しているにすぎない。記述的な指示文の使用に対していかなる診断メッセージを出すことも、この言語の移植性のある機能ではないことに注意されたい。特に、再マッピングの必要性がないにもかかわらず、コンパイラがこのこと実を確かめるのが不可能か又は自明でない場合があることに注意が必要である。コンパイラが異なれば、違った状況でメッセージを出力する場合もある。さらに、そのようなメッセージを出力しなければならないという要求も全くない。【以上】

記述的な指示文は、マッピングの記述の前にアスタリスクが付くことを除いては、指令的な指示文と同じ形をしている。

```
!HPF$ DISTRIBUTE A *(BLOCK, CYCLIC)
```

は、記述的な指示文である。

**転写的** マッピングは、指定されない。呼ばれた側の副プログラムは、渡ってきた引数がどのようにマップされていてもそれを受け入れる必要がある。もちろんこのことは、呼出し元 (の実装) がマッピングに関する情報を実行時に渡すことを意味する。

転写的な指示文は、分散形式及びプロセッサ構成を指定するところに単にアスタリスクを書くことで記述できる。

```
!HPF$ DISTRIBUTE A *
!HPF$ DISTRIBUTE B * ONTO *
```

は、転写的な指示文である。転写的な整列を指定するのには、INHERIT 指示文 (第 4.4.2 項を参照) が用いられる。

分散形式と、プロセッサ構成の両方に、指令的、記述的、複写的の指示ができる。整列は、より複雑である。なぜなら、仮引数が整列すべきテンプレートが必要だからである。このテンプレートは、指定しなくても構わない (この場合もちろん ALIGN 指示文は存在しない)。この時、仮引数のテンプレートは、**自然テンプレート** になる。(“自然テンプレート” は、この後の第 4.4.1 項で定義される。) そうでない場合は、次のそれぞれの項目の内の一つが成り立たなければならない。

- テンプレートは、指令的な ALIGN 指示文により明示的に指定される
- テンプレートは、記述的な ALIGN 指示文により明示的に指定される
- テンプレートは、継承される。これは、仮引数に INHERIT 属性 (この後の第 4.4.2 項で説明されている) を与えることで指定される。これは、テンプレートが、対応する実引数が最終的に整列されているテンプレートのコピーとなることを暗黙に指定する。さらに、仮引数のそのテンプレートへの整列は、対応する実引数のそれと同じになる。これは結局、整列の転写的な形式と同じ効果を持つ。

これについては、この後の第 4.4.1 項でより正確に説明されている。

【利用者への助言】 例えば、一部分が指令的で、一部分が転写的であるような混合したマッピング指示文を記述することは可能だが、そうすることにはおそらく何のメリットもないだろう。これらの指示文のポイントは、コンパイラが、必要な再マッピングをすべて正しく、効率的に処理できるようにするところにある。再マッピングは、次のような理由が原因で起こることがある。

- 実引数と仮引数の整列状態を合わせるため
- 実引数と仮引数の分散状態を合わせるため
- 実引数と仮引数が分散されるプロセッサ構成を一致させるため

ほとんどのマシンにとって、これらのどの理由による再マッピングの間にも、実質的なコストの差はない。それゆえ、マッピング指示を純粋に転写的な指示にする、純粋に指令的な指示にする、あるいは純粋に記述的な指示にするほうが、よい選択といえるであろう (少なくともそのほうが読みやすくなる)。

転写的な指示によるマッピングは、ライブラリを書く場合に有用かもしれないが、副プログラム中でランタイムコストを発生することになる。それゆえ、通常のユーザーコード中でこれらを使用することは避けるべきである。【以上】

## 4.2 どのような再マッピングが必要とされ、誰がそれを行なうのか

呼び出される副プログラムに対する明示的引用仕様が宣言されており、その引用仕様が仮引数に対する指令的又は記述的なマッピング指示文を含み、かつ対応する実引数の再マッピングが必要な場合、その副プログラムの呼出しは、データが、明示された引用仕様で指示された仮引数のマッピングにマッチするようなテンポラリの変数に、あたかもコピーされたかのようにして行なわれる。仮引数のテンプレートは、引用仕様に記述された通りになる。

明示的引用仕様が存在しない場合は、再マッピングは必要ない。これは、第 4.6 節で示される要求の結果である。

簡単のため、この章で示されるコード片には、そのように必要とされる明示的引用仕様が全て含まれているわけではないことに注意されたい。

基本となる原理は、**引数の再マッピングは、呼出し元からは見えない**ということである。つまり、副プログラムからの戻り時や、呼出し元の実行が再開された時には、その呼出し後に呼出し元がアクセスできる全ての実体は、呼出し前と全く同様にマップされているという

ことである。手続は、それを呼び出した側からも見えるようないかなる実体のマッピングを変更することもできない。

⇓ 【利用者への助言】いくつかの公認拡張では、この制約が緩和される。例えば、第 8.6 節や第 8.8 節を参照してほしい。【以上】

## 4.3 分散とプロセッサ構成

全ての *distributee* が仮引数であるような DISTRIBUTE 指示文では、*dist-format-clause* か *dist-target* あるいはその両方を、アスタリスクで始めたり、アスタリスクだけから構成したりできる。

- アスタリスクがない場合、*dist-format-clause* 又は *dist-target* は、指令的になる。この節は分散を記述し、言語処理系にその分散に従うよう要求する。これによって、要求された仮引数の分散を実現するために、実行時に実引数の再マッピング又はコピーが、呼出し元か呼ばれた側の副プログラム (の実装) で行なわれる可能性がある。
- アスタリスクで始めると、*dist-format-clause* 又は *dist-target* は、記述的になる。この指示文は、次の点を除いて、指令的な指示文と全ての点において同等である。すなわち、コンパイラが、実引数の再マッピングが必要ないということが判定できなかった場合、そのことについての診断メッセージを出力できるということである。この点についての詳しい説明は、第 4.1 節を参照してほしい。
- アスタリスクのみからなると、*dist-format-clause* 又は *dist-target* は、転写的になる。この節は、分散については何も述べないが、言語処理系に対して、実引数の分散をコピーすることを要求する。(この意図は、引数が参照渡しの場合、実行時にデータの移動が不要になるということである。)

一つの DISTRIBUTE 指示文の中で、*dist-format-clause* がアスタリスクを持ち、*dist-target* が持たなくてもよく、またその逆も可能である。

### 4.3.1 例

以下の仮引数用の DISTRIBUTE 指示文の例は、いろいろな組合せについて説明している。

```
!HPF$ DISTRIBUTE URANIA (CYCLIC) ONTO GALILEO
```

言語処理系は、プロセッサ構成 GALILEO へ URANIA を CYCLIC 分散するために必要なことをしなければならない。

```
!HPF$ DISTRIBUTE POLYHYMNIA * ONTO ELVIS
```

言語処理系は、POLYHYMNIA を、その現在の分散形式 (それが何であれ) に従って、プロセッサ構成 ELVIS へ分散するために必要なことをしなければならない。(POLYHYMNIA は他のプロセッサ構成の可能性もある。)

```
!HPF$ DISTRIBUTE THALIA *(CYCLIC) ONTO *FLIP
```

言語処理系は、プロセッサ構成 FLIP へ THALIA を CYCLIC に分散するために必要なことをしなければならない。プログラマは、実引数がすでにそのように分散されており、再マッピングは必要ないことを知っている。

```
!HPF$ DISTRIBUTE EUTERPE (CYCLIC) ONTO *
```

言語処理系は、実引数が分散されていたプロセッサ構成上に EUTERPE を CYCLIC 分散するために、必要なことをしなければならない。

```
!HPF$ DISTRIBUTE ERATO * ONTO *
```

ERATO のマッピングは、実引数のマッピングから変更してはいけない。

DISTRIBUTE ERATO * ONTO *は、

```
!HPF$ DISTRIBUTE ERATO (*) ONTO *
```

とは同じ意味を持たないことに注意されたい。後者は、ERATO を実引数が分散されていたプロセッサ構成上に、*(つまり、オンプロセッサ) 分散するという意味になる。この場合は、プロセッサ構成は必然的にスカラになる。

### 4.3.2 省略時の解釈

仮引数については、*dist-format-clause* 又は *dist-onto-clause* のどちらか一方を省略しても構わない。これは、次のように解釈される。

仮引数が INHERIT 属性 (第 4.4.2 項参照) を持っている時は、いかなる場合においても分散指示をすることは許されない。分散も、整列と同じく実引数から引き継がれる。

その他のケースで、分散情報が省略されている場合、コンパイラは任意に分散形式又はターゲットとなるプロセッサ構成を選択できる。

以下に 2 つの例を示す。

```
!HPF$ DISTRIBUTE WHEEL_OF_FORTUNE *(CYCLIC)
```

プログラマは、WHEEL_OF_FORTUNE 仮引数に対応する実引数がすでに CYCLIC 分散されていることを知っている。コンパイラは、渡ってきたデータが本当に CYCLIC かどうか確かめ、もしそうでない場合は、必要ならば再マッピングを行なわなければならない。データは、何か他のプロセッサ構成に再マップすることもできるが、そうする理由はない。もしそのような再マッピングを行った場合は、プログラマを驚かせることになるであろう。

```
!HPF$ DISTRIBUTE ONTO *TV :: DAVID_LETTERMAN
```

プログラマは、DAVID_LETTERMAN 実引数に対応する仮引数が、TV に、何らかの方法で分散されていることを知っている。コンパイラは、それが本当かどうか確かめ、本当でないなら、そうなるようにする必要がある。分散形式は、DAVID_LETTERMAN が TV 上に保持される限り、変更できる。(この宣言は、属性形式で記述されなければならないことに注意されたい。

```
!HPF$ DISTRIBUTE DAVID_LETTERMAN ONTO *TV          ! 規格合致でない
```

という文形式は、DISTRIBUTE 指示文の構文に適合しない。)

## 4.4 整列

### 4.4.1 仮引数のテンプレート

ここで、仮引数が最終的に整列されているテンプレートがどのように決められるのかについて詳細に述べる。

テンプレートは、副プログラムの引数引用仕様を通して渡されることはない。仮引数と、それに対応する実引数が同じテンプレートに整列するのは、テンプレートが、親子結合や参照結合により、呼出し元と呼ばれた副プログラム側の両方からアクセス可能な場合のみである。その他の場合は、仮引数が整列するテンプレートと実引数が整列するテンプレートは常に異なる。ただし、テンプレートがコピーされたものである場合はあり得る (第 4.4.2 項参照)。HPF の実装では、手続からの戻りの際に、実引数が、手続呼出しの前に整列していたテンプレートと同じテンプレートに整列される。

仮引数のテンプレートは、次の三つのいずれかの方法で作成される。

- 仮引数が `ALIGN` 指示文の中で *alignee* として明示的に使用されている場合、*align-target* がテンプレートであれば、仮引数のテンプレートはその *align-target* である。そうでなければ、仮引数のテンプレートは、*align-target* が最終的に整列されているテンプレートである。

- 仮引数が明示的に整列されておらず、しかも `INHERIT` 属性 (第 4.4.2 項で説明されている) を持っていない場合、そのテンプレートは、仮引数と同じ形状と上下限を持ち、その仮引数の自然テンプレートと呼ばれる。

  (よって、第 4.3 節の全ての例では、自然テンプレートが使用されていることになる。)

- 仮引数が明示的に整列されておらず、しかも `INHERIT` 属性を持っている場合、そのテンプレートは次の規則にしたがって実引数から「継承」される。

  – 実引数が全体配列である場合、仮引数のテンプレートは、実引数が最終的に整列されているテンプレートのコピーである。
  – 実引数がどの添字もベクトル添字でないような配列 $A$ の部分配列である場合、仮引数のテンプレートは、$A$ が最終的に整列するテンプレートのコピーである。
  – 実引数がその他の式である場合、テンプレートの形状と分散は、言語処理系が自由に選択できる（したがって、プログラマはその分散について前もって何も知ることはできない)。

  これらすべての場合において、仮引数は継承 (`inherited`) テンプレートを持つと言う。

### 4.4.2 INHERIT 指示文

`INHERIT` 指示文は、仮引数を、対応する実引数のテンプレートのコピーへ、実引数が整列されているのと同じ方法で整列することを指定する。

H401 *inherit-directive* **is** `INHERIT` *inheritee-list*

H402 *inheritee* **is** *object-name*

制約: *inheritee* は仮引数でなければならない。

制約: *inheritee* は *alignee* であってはならない。

制約: *inheritee* は *distributee* であってはならない。

> 【利用者への助言】 公認拡張では、上記の 3 つの制約の内の一つ目は、ポイントに対しては緩和される (第 8.8 節を参照してほしい)。【以上】 ⇓

INHERIT 指示文は、指定した副プログラムの仮引数に INHERIT 属性を持たせる。INHERIT 属性を持てるのは仮引数だけである。ある実体が INHERIT 属性と ALIGN 属性の両方を持つことはできない。ある実体が INHERIT 属性と DISTRIBUTE 属性の両方を持つことはできない。INHERIT 指示文は、有効域の *specification-part* の中でだけ使用できる。

INHERIT 属性は、仮引数のテンプレートが、実引数のテンプレートのコピーを作成することによって継承されたものでなければならないことを指定する。さらに、INHERIT 属性を指定された引数は、他の明示的なマッピング指示文を指定することはできない。INHERIT 属性は、継承テンプレートに対する DISTRIBUTE * ONTO * の分散を意味する。ゆえに、最終的な効果は、コンパイラに、データを正確にそのままの位置にとどめ、実引数を再マップしてはならないということを伝えることにある。仮引数は、実引数と全く同じ方法でマップされる。副プログラムは、実引数が論理プロセッサにどのようにマップされていても、正しく動くようにコンパイルされなければならない。

A が、配列仮引数である場合、

```
!HPF$ INHERIT A
```

という指示文は、

```
!HPF$ DISTRIBUTE A * ONTO *
```

よりもより一般的であることに注意されたい。その理由は次の通りである。INHERIT 指示文は、A が整列している継承テンプレートが、* ONTO*で分散されるが、A はそのテンプレートに、何らかの自明でない方法で整列するということを意味する。一方、DISTRIBUTE 指示文は、A は明白にその自然テンプレートに整列し、その後* ONTO*で分散されるという意味になる。

例えば、次のようなコードは許されない。

```
!HPF$ PROCESSORS P(2)
      REAL, DIMENSION(100) :: A
!HPF$ DISTRIBUTE (BLOCK) ONTO P :: A

      CALL FOO(A(1:50))


      ...

      SUBROUTINE FOO(D)
      REAL, DIMENSION(50) :: D
!HPF$ DISTRIBUTE D *            ! 規格合致でない
```

D に対する転写的な指示文は、D の自然テンプレートが BLOCK 分散されていないため、規格合致していない。一方、この間違った指示文を次の指示文で置き換えることは、正当な処置である。

```
!HPF$ INHERIT D
```

#### 4.4.2.1 例

ここで、INHERIT の使用法について簡単な例をあげる。

```
      REAL DOUGH(100)
!HPF$ DISTRIBUTE DOUGH(BLOCK(10))
      CALL PROBATE( DOUGH(7:23:2) )
      ...
      SUBROUTINE PROBATE(BREAD)
      REAL BREAD(9)
!HPF$ INHERIT BREAD
```

BREAD の継承テンプレートは形状 [100] を持つ。要素 BREAD(I) は継承テンプレートの要素 5 + 2*I に整列し、そのテンプレートは BLOCK(10) の分散を持つ。

もっと複雑な例も簡単に作ることができる。継承テンプレートは、仮引数の次元数とは異なった次元数を持つことができ、さらにそれは実引数の次元数とも異なった次元数を持ってもよいということを気にとめておくことは重要である。例えば、次のような部分がプログラムに含まれていたとする。

```
      REAL A(100,100)
!HPF$ TEMPLATE T(100,100,100)
!HPF$ DISTRIBUTE T(BLOCK,CYCLIC,*)
!HPF$ ALIGN A(I,J) with T(J,3,I)
      CALL SUBR(A(:,7))
      ...
      SUBROUTINE SUBR(D)
      REAL D(100)
!HPF$ INHERIT D
```

この場合、仮引数 D は 1 次元である。D は、2 次元の実引数 A の 1 次元部分に対応し、A は、3 次元テンプレート T の 2 次元断面に整列する。D のテンプレートは、この 3 次元テンプレートのコピーである。D は、この継承テンプレートの (7, 3, :) の部分に整列する。よって、仮引数 D の「見える」次元は、*に分散される。しかし、例えばもし CALL 文が、

```
      CALL SUBR(A(7,:))
```

の形だった場合、仮引数の「見える」次元は、BLOCK に分散される。

### 4.4.3 記述的な ALIGN 指示文

*align-spec* の先頭にアスタリスクがあるかないかは、*dist-format-clause* の場合と同じ意味を持つ。つまり、アスタリスクがあるかないかは、ALIGN 指示文が記述的なのか指令的なのかをそれぞれ指定する。

*で始まらない *align-spec* が仮引数に適用された場合、その意味は、その仮引数が副プログラムへの入口で強制的に指定の整列を持たされるということである。この場合、呼出し元かもしくは副プログラム側 (の実装) で、実引数のデータ又はそのコピーの一時的な再マッピングを必要とする場合がある。

仮引数は、*align-target* としても使用できることに注意されたい。

```
      SUBROUTINE NICHOLAS(TSAR,CZAR)
      REAL, DIMENSION(1918) :: TSAR,CZAR
!HPF$ INHERIT :: TSAR
!HPF$ ALIGN WITH TSAR :: CZAR
```

この例では、最初の仮引数 TSAR は対応する実引数に整列したままとなるが、2 番目の仮引数の CZAR は強制的に最初の仮引数へ整列させられる。二つの実引数がすでに整列していれば、実行時にデータの再マッピングは必要ない。そうでなければ、何らかの再マッピングが起きるであろう。

*align-spec* が “*” で始まる場合、*alignee* は仮引数でなければならない。“*” は、実引数がすでに指定された整列をしており、実行時にそれを再マップする必要はないということを、プログラマが知っているということを示している。(前に述べた通り、このプログラマの知識が正しくなければならないという要求はない。コンパイラは、指令的な整列の場合と同じく、必要と思われるならば、再マッピングを生成しなければならない。) 例えば、上の例で、ALIGN 指示が次のように変更されたとする。

```
!HPF$ ALIGN WITH *TSAR :: CZAR
```

これは、プログラマが、TSAR に対応する実引数の再マッピングは必要ないということを表明していることになる。

単に “ALIGN WITH *” という言い方は許されない。アスタリスクの後に *align-target* がなければならない。(「どの整列でも受け入れる」と言う正しい方法は、INHERIT である。)

仮引数が明示的な ALIGN 属性又は DISTRIBUTE 属性を持たない場合、コンパイラは暗黙の整列と分散を指定する。それは、「表明のアスタリスク」なしに明示的に記述できるものの中から選択される。

#### 4.4.3.1 例

INHERIT を使用しない場合、副プログラム境界で再マッピングが起きないことを保証するには、仮引数の明示的な整列が必要となるであろう。次の例を考えてもらいたい。

```
      LOGICAL FRUG(128)
!HPF$ PROCESSORS DANCE_FLOOR(16)
!HPF$ DISTRIBUTE (BLOCK) ONTO DANCE_FLOOR::FRUG
      CALL TERPSICHORE(FRUG(1:40:3))
```

部分配列 FRUG(1:40:3) は、次のような方法で論理プロセッサへマップされる。

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | | | 25 | | | | | | | | | | | | |
| | 10 | | | 34 | | | | | | | | | | | |
| | | 19 | | | | | | | | | | | | | |
| 4 | | | 28 | | | | | | | | | | | | |
| | 13 | | | 37 | | | | | | | | | | | |
| | | 22 | | | | | | | | | | | | | |
| 7 | | | 31 | | | | | | | | | | | | |
| | 16 | | | 40 | | | | | | | | | | | |

まず、サブルーチン TERPSICHORE の引用仕様が次のようだったとする。

```
      SUBROUTINE TERPSICHORE(FOXTROT)
      LOGICAL FOXTROT(:)
!HPF$ INHERIT FOXTROT
```

FOXTROT のテンプレートは、全体配列 FRUG の 128 要素からなるテンプレートのコピーである。このテンプレートは次のようにマップされる。

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 9 | 17 | 25 | 33 | 41 | 49 | 57 | 65 | 73 | 81 | 89 | 97 | 105 | 113 | 121 |
| 2 | 10 | 18 | 26 | 34 | 42 | 50 | 58 | 66 | 74 | 82 | 90 | 98 | 106 | 114 | 122 |
| 3 | 11 | 19 | 27 | 35 | 43 | 51 | 59 | 67 | 75 | 83 | 91 | 99 | 107 | 115 | 123 |
| 4 | 12 | 20 | 28 | 36 | 44 | 52 | 60 | 68 | 76 | 84 | 92 | 100 | 108 | 116 | 124 |
| 5 | 13 | 21 | 29 | 37 | 45 | 53 | 61 | 69 | 77 | 85 | 93 | 101 | 109 | 117 | 125 |
| 6 | 14 | 22 | 30 | 38 | 46 | 54 | 62 | 70 | 78 | 86 | 94 | 102 | 110 | 118 | 126 |
| 7 | 15 | 23 | 31 | 39 | 47 | 55 | 63 | 71 | 79 | 87 | 95 | 103 | 111 | 119 | 127 |
| 8 | 16 | 24 | 32 | 40 | 48 | 56 | 64 | 72 | 80 | 88 | 96 | 104 | 112 | 120 | 128 |

FOXTROT(I) は、このテンプレートの要素 3*I-2 に整列する。

一方、TERPSICHORE の引用仕様が、代わりに次のようだったと仮定する。

```
      SUBROUTINE TERPSICHORE(FOXTROT)
      LOGICAL FOXTROT(:)
!HPF$ DISTRIBUTE FOXTROT(BLOCK)
```

この場合、FOXTROT のテンプレートは、FOXTROT 自身と同じ大きさ 14 の自然テンプレートである。実引数 FRUG(1:40:3) は、次のような方法で 16 のプロセッサにマップされる。

| Abstract processor | Elements of FRUG |
|---|---|
| 1 | 1, 2, 3 |
| 2 | 4, 5, 6 |
| 3 | 7, 8 |
| 4 | 9, 10, 11 |
| 5 | 12, 13, 14 |
| 6–16 | none |

すなわち、仮引数の要素の (実引数のテンプレート上での) 位置は、次のようになる。

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | | | 9 | | | | | | | | | | | | |
| | 4 | | | 12 | | | | | | | | | | | |
| | | 7 | | | | | | | | | | | | | |
| 2 | | | 10 | | | | | | | | | | | | |
| | 5 | | | 13 | | | | | | | | | | | |
| | | 8 | | | | | | | | | | | | | |
| 3 | | | 11 | | | | | | | | | | | | |
| | 6 | | | 14 | | | | | | | | | | | |

このようなレイアウト (最初のプロセッサに 3 要素、2 番目に 3 要素、3 番目に 2 要素、4 番目に 3 要素、...) は、BLOCK 分散として適切に表すことはできない。それゆえ、呼出しの際に再マッピングが起きるであろう。

仮引数を、大きさ 128 の BLOCK 分散されたテンプレートを宣言し、それに明示的に整列すれば、INHERIT を使わずに、再びマップを防ぐことができる。

```
      SUBROUTINE TERPSICHORE(FOXTROT)
      LOGICAL FOXTROT(:)
!HPF$ PROCESSORS DANCE_FLOOR(16)
!HPF$ TEMPLATE, DISTRIBUTE(BLOCK) ONTO DANCE_FLOOR::GURF(128)
!HPF$ ALIGN FOXTROT(I) WITH GURF(3*I-2)
```

【利用者への助言】後者のテクニックの利点は、それが使用できるところであれば、コンパイラにより多くの情報を与えられるという点である。この情報は、より効率的なコードを生成するのに使用することができる。【以上】

## 4.5 マッピングの集合における同値関係と半順序

名前付き実体のマッピングの集合は、ある種の同値関係を法とする半順序を持つ。おおまかにいうと、$P$ と $Q$ が二つのマッピングである時、$Q$ が $P$ の「特殊化」であるということ (すなわち、この順序関係において「$Q$ が $P$ の下にある」ということ) は、$P$ が部分的に指定さ

れており、$Q$ は、$P$ を満たすようなマッピングの一つであるということである。この記法は、この後の第 4.6 節と　第 8.8 節において用いられる。

【利用者への助言】 これらの条件に対する説明は非常に複雑なので、もしあなたが常に明示的引用仕様を指定し (この後で説明されるように、これは非常に簡単で、たいていの場合自動的にできる)、マッピングされたポインタ (第 8.8 節で説明されている承認された拡張である) を使わないならば、この章を読み飛ばして構わない。【以上】

⇓

詳細な定義を以下に示す。
まず、*dist-format* 指定子の同値性について定義する。

1. 「同値である」という表現に記法 $\equiv$ を用いると、

$$
\begin{aligned}
\texttt{BLOCK} &\equiv \texttt{BLOCK} \\
\texttt{CYCLIC} &\equiv \texttt{CYCLIC} \\
* &\equiv * \\
\texttt{BLOCK}(n) &\equiv \texttt{BLOCK}(m) \quad ! \, m \text{ と } n \text{ の値が等しいとき、かつそのときに限り} \\
\texttt{CYCLIC}(n) &\equiv \texttt{CYCLIC}(m) \quad ! \, m \text{ と } n \text{ の値が等しいとき、かつそのときに限り} \\
\texttt{CYCLIC} &\equiv \texttt{CYCLIC(1)}
\end{aligned}
$$

2. これ以外の、字面上異なる *dist-format* 指定は、同値ではない。

これは、一般の数学的な意味での同値関係と同じである。

次に、マッピングの半順序を定義する。S (“special”) と G (“general”) を二つのデータ実体とする。

次のいずれかの条件が成り立つ場合、かつその場合に限り、S のマッピングは、G のマッピングの**特殊化**であるという。

1. G は INHERIT 属性を持つ
2. S は INHERIT 属性を持たず、次の全ての条件が成り立つ
   (a) S は名前付き実体である
   (b) S と G の最終整列先が同じ形状を持つ。かつ、
   (c) S と G の対応する次元が、それぞれの最終整列先の対応する次元にマッピングされ、S と G の対応する要素が、それぞれの最終整列先の対応する同じ要素に整列する。かつ、
   (d) 次のいずれかが成り立つ
      i. S と G の最終整列先が明示的に分散されていない。または、
      ii. S と G の最終整列先が両方とも明示的に分散されている。この場合、G の最終整列先に対する分散指示文は、次の条件の内の一つを満たさなければならない。
         A. *dist-onto-clause* を持たない。または、
         B. *dist-onto-clause* が “ONTO *” である。または、
         C. S の最終整列先に対して明示的に指定された分散指示文で指定されたのと同じ形状を持つプロセッサ構成を指定する *dist-onto-clause* を持つ

      さらに次の条件の内の一つを満たさなければならない。
         A. *dist-format-clause* を持たない。または、

B. *dist-format-clause* が “*” である。または、
C. 各 *dist-format* は、S の最終整列先に対する明示的分散指示文の *dist-format-clause* の対応する位置の *dist-format* と (上で定義された意味で) 同値である。

この定義によれば、以下のことがいえる。

- 名前付き実体に対する任意のマッピングは、それ自身の特殊化になっている
- $A$、$B$、$C$ が名前付き実体で、$A$ が $B$ のマッピングの特殊化になっており、$B$ が $C$ のマッピングの特殊化になっている場合、$A$ は $C$ のマッピングの特殊化になっている

すなわち、名前付き実体に適用される特殊化の関係は反射的かつ推移的であり、それゆえその関係は、名前付き実体のマッピング集合に対する同値関係を作るのに使用できる。つまり、お互いに特殊化の関係にある二つのマッピングは、同値関係にあるということができる。この定義によれば、特殊化の関係が、名前付き実体のマッピング集合に対する、同値関係を法とする半順序を生じさせる。INHERIT マッピングは、この半順序の中の唯一の極大元である。

## 4.6 明示的引用仕様を省略するための条件

ある条件では、副プログラムの明示的引用仕様は必要ない。マッピング指示を使用した HPF プログラムでは、Fortran において許容された引用仕様の省略の条件はかなり厳しくなる。

【利用者への助言】 これらの条件は複雑である。明示的引用仕様を使用していれば、本節を読む必要はないことを是非承知しておいてほしい。もし何等かの疑問が浮かんだならば、すぐに明示的引用仕様を使用しているか確認してもらいたい。【以上】

以下の全ての条件が満たされている場合**以外は**、明示的引用仕様が必要である。

1. Fortran 規格によって必要とされていないこと。**かつ**
2. 仮引数は、転写的な分散が指示されてなく、INHERIT 属性を持たないこと。**かつ**
3. それぞれの対応する実引数と仮引数は、
   (a) どちらも暗黙のマップされているか、
   (b) どちらも明示的にマップされており、実引数のマッピングが仮引数のマッピングの特殊化になっている。

   **かつ、**
4. それぞれの対応する実引数と仮引数は、以下のいずれかであること。
   (a) ともに順序的であるか、
   (b) ともに非順序的である。

【仕様の根拠】 これは、以下のような結果をもたらす。

- 純粋な (すなわち、HPF 指示文がない)Fortran プログラムは、少なくとも全ての変数がデフォルトでは順序的であるようなコンパイル環境においては、引用仕様を追加しなくても HPF 規格合致でありつづける。これは、項目 1、2、3(a)、および 4(a) によって保証される。

- もし再マッピングが必要ならば、呼出し元はその事実を認識できるであろう。従って実装では全ての再マッピングを呼出し元で実行するよう選択することができる。

【以上】

【利用者への助言】 この要請はユーザに明示的引用仕様を常に記述するよう、強く求めている。これは良いことであり、明示的引用仕様によって多くの誤りをコンパイル時に検出でき、頑健なソフトウェアの開発過程を大いにスピードアップすることができる。

明示的引用仕様は 3 通りの方法で規定されうることに注意されたい。

1. モジュール副プログラムが明示的引用仕様を持つ場合。
2. 内部副プログラムが明示的引用仕様を持つ場合。
3. 明示的引用仕様が引用仕様宣言によって規定される場合。

更に、組込み手続は定義によりいつも明示的引用仕様を持つ。

慣用的な Fortran のプログラミング手法では、モジュールの使用法は広範なものになっている。例えば、どの副プログラムもモジュールに含むことができる。この手法では、プログラマが特別に努力することなく、明示的引用仕様が自動的に提供される。引用仕様宣言を記述する必要があるのは非常に希なはずである。【以上】

## 4.7 手続の特性

Fortran 規格 (F95:12.2) に与えられる仮データ実体や関数結果の特性は、実体の ***HPF* 特性** (*hpf-characreristics*) をも含むように拡張される。これは、以下のように再帰的に定義される。

- プロセッサ構成は、一つの HPF 特性を持つ。その形状である。
- テンプレートは以下の 3 つの HPF 特性を持つ。
  1. その形状。
  2. 明示的に記述されている場合、その分散。
  3. 明示的に記述されている場合、そのテンプレートが分散されているプロセッサ構成の HPF 特性 (すなわち形状)。
- 仮データ実体は、以下の HPF 特性を持つ。
  1. 明示的に指定されている場合、その整列。およびその整列先の全ての HPF 特性。
  2. 明示的に指定されている場合、その分散。および明示的に指定されている場合、その仮データ実体が分散されているプロセッサ構成の HPF 特性 (すなわち形状)。
- 関数の結果は仮データ実体と同じ HPF 特性を持つ。特に、以下の HPF 特性を持つ。
  1. 明示的に指定されている場合、その整列。およびその整列先の全ての HPF 特性。
  2. 明示的に指定されている場合、その分散。および明示的に指定されている場合、その関数結果が分散されているプロセッサ構成の HPF 特性 (すなわち形状)。

【仕様の根拠】 明示的引用仕様が引用仕様宣言によって与えられた場合、Fortran 規格では、当該引用仕様宣言にどのような情報を指定しなければならないかを規定して

いる。その規定は、Fortran の**特性**の概念を使用して行なわれる。例えば、仮データ実体の特性には、その型が含まれる。特性は引用仕様宣言中に指定しなければならない。Fortran 規格 (F95:12.3.2.1) で、

> 引用仕様本体は、手続の特性のすべてを指定する。これらは、手続の定義と一致していなければならない...

と、規定されている。通常、手続の引用仕様宣言は当該手続内の該当する宣言の字面どおりのコピーである。この章では、単に、そのコピーが手続の仮引数に関する全ての明示的なマッピングの指示を含んでいなければならないと述べているだけである。【以上】

## 4.8 引数の受け渡しと順序結合

手続呼出しにおける実引数について、Fortran は配列要素 (スカラ) と結合する仮引数が配列であることを許している。更に、仮引数の形状を対応する実引数と異なるものにすることができ、その結果、手続呼出しを通して実引数の形状を変更することも許している。Fortran の記憶列の性質が、仮引数の値を定めるために使われてきた。Fortran 77 から継承されたこの機能は、より大きな配列の部分行または列の開始アドレスを手続に渡すために広く使用されてきた。HPF の配列はプロセッサにまたがってマッピングされる可能性があるため、この機能は完全にはサポートされていない。

### 4.8.1 順序結合規則

1. 配列要素または大きさ引継ぎ配列の名前を、実引数として使用する場合は、結合する仮引数はスカラであるか、順序的配列の指定がなければならない。
   非順序的配列の配列要素特定子は、配列仮引数と結合してはならない。
2. 実引数が配列または部分配列であり、対応する仮引数と形状が異なる場合は、仮引数は順序的であると宣言されており、実配列引数は順序的でなければならない。
3. (スカラまたは配列の) 文字型の実体は、もしそれが第 3.8.1.1 項の定義 4 を満たしているなら、非順序的である。長さ明示の (explicit-length) 文字型仮引数の長さが、実引数の長さと異なる場合、仮引数と実引数はともに順序的でなければならない。
4. 明示的引用仕様がないとき、順序的実引数は非順序的仮引数と結合することはできない。また、非順序的実引数は順序的仮引数と結合することはできない。(この項目は第 4.6 節の一部の繰り返しにすぎない。)

### 4.8.2 順序結合についての説明

配列仮引数とそれに結合する配列実引数との形状が異なるとき、実引数は式 (expression) であってはならない。配列値を持つ式 (array-valued expression) が順序的であることを宣言する HPF のメカニズムは存在しない。その様な式を実引数として次元数の異なる仮引数と結合するためには、その実引数を、第 3.8.1.1 項の定義 4 に従って順序的であるよう強制された名前つき配列変数に代入する必要がある。

### 4.8.3 順序結合の例

以下のようなサブルーチンの一部が与えられたとき、

```
SUBROUTINE HOME (X)
DIMENSION X (20,10)
```

規則 1 により、

```
CALL HOME (ET (2,1))
```

という記述は、X が HOME において順序的であると宣言されており、ET が呼出し元の手続内で順序的であるときだけ正しいものである。

同様に、規則 2 及び 4 により、

```
CALL HOME (ET)
```

という記述は、ET および X が、ともに順序的配列であるか、ET および X が同じ形状を持っており、(明示的引用仕様がない場合には) 同じ順序属性を持っているかのどちらかであることを要請する。

規則 3 は、文字型実体に対して特別な考慮を示している。以下のような手続呼出しにまたがった文字型実体の長さの変化は、Fortran の概念では正しいものである。

```
CHARACTER (LEN=44) one_long_word
one_long_word = 'Chargoggagoggmanchaugagoggchaubunagungamaugg'
CALL webster(one_long_word)

SUBROUTINE webster(short_dictionary)
CHARACTER (LEN=4) short_dictionary (11)
    !例えば、short_dictionary(3) は 'agog'となることに注意
```

HPF においては、実引数と仮引数はともに順序的でなければならない。(例の中の Chargoggagoggmanchaugagoggchaubunagungamaugg は、現在は「ウェブスター湖 (Lake Webster)」と呼ばれているマサチューセッツ州の湖にニプマク族がつけた元の名前である)。