Section 4

Data Parallel Statements and Directives

The purpose of the FORALL statement and construct is to provide a convenient syntax for simultaneous assignments to large groups of array elements. Such assignments lie at the heart of the data parallel computations that HPF is designed to express. The multiple assignment functionality it provides is very similar to that provided by the array assignment statement and the WHERE construct in Fortran 90. FORALL differs from these constructs in its syntax, which is intended to be more suggestive of local operations on each element of an array, and in its generality, which allows a larger class of array sections to be specified. In addition, a FORALL may call user-defined functions on the elements of an array, simulating Fortran 90 elemental function invocation (albeit with a different syntax).

HPF defines a new procedure attribute, PURE, to declare the class of functions that may be invoked in this way. Both single-statement and block FORALL forms are defined in this Section, as well as the PURE attribute and constraints arising from the use of PURE.

HPF also defines a new directive, INDEPENDENT. The purpose of the INDEPENDENT directive is to allow the programmer to give additional information to the compiler. The user can assert that no data object is defined by one iteration of a DO loop and used (read or written) by another; similar information can be provided about the combinations of index values in a FORALL statement or construct. Such information is sometimes valuable to enable compiler optimizations, but may require knowledge of the application that is available only to the programmer. Therefore, HPF allows a user to specify these assertions, on which the compiler may in turn rely in its translation process. If the assertion is true, the semantics of the program are not changed; if it is false, the program is not HPF-conforming and has no defined meaning.

4.1 The FORALL Statement

Fortran 90 places several restrictions on array assignments. In particular, it requires that operands of the right side expressions be conformable with the left hand side array. These restrictions can be relaxed by introducing the element array assignment statement, usually referred to as the FORALL statement. This statement is used to specify an array assignment in terms of array elements or groups of array sections, possibly masked with a scalar logical expression. In functionality, it is similar to array assignment statements and WHERE statements. The FORALL statement essentially preserves the semantics of Fortran 90 array

assignments and allows for convenient assignments like

```
FORALL ( i=1:n, j=1:m ) a(i,j)=i+j
```

as opposed to standard Fortran 90

```
a = SPREAD((/(i,i=1,n)/), DIM=2, NCOPIES=m) +     &
    SPREAD((/(i,i=1,m)/), DIM=1, NCOPIES=n)
```

It can also express more general array sections than the standard triplet notation for array expressions. For example,

```
FORALL ( i = 1:n ) a(i,i) = b(i)
```

assigns to the elements on the main diagonal of array a.

Rationale. It is important to note, however, that FORALL is not intended to be a general parallel construct; for example, it does not express pipelined computations or MIMD computation well. This was an explicit design decision made in order to simplify the construct and promote agreement on the statement's semantics. (*End of rationale.*)

4.1.1 General Form of Element Array Assignment

Rule R215 in the Fortran 90 standard for *executable-construct* is extended to include the *forall-stmt.*

H401 *forall-stmt* **is** FORALL *forall-header forall-assignment*

H402 *forall-header* **is** (*forall-triplet-spec-list* [, *scalar-mask-expr*])

Constraint: Any procedure referenced in the *scalar-mask-expr* of a *forall-header* must be pure, as defined in Section 4.3.

Rationale. Pure functions are guaranteed to be free of side effects. Therefore, they are safe to invoke in the *scalar-mask-expr.*

Note that functions referenced in the *forall-triplet-spec-list* are not syntactically constrained as the *scalar-mask-expr* is. This is consistent with the handling of bounds expressions in DO loops. (*End of rationale.*)

H403 *forall-triplet-spec* **is** *index-name* = *subscript* : *subscript* [: *stride*]

Constraint: *index-name* must be a scalar integer variable.

Constraint: A *subscript* or *stride* in a *forall-triplet-spec-list* must not contain a reference to any *index-name* in the *forall-triplet-spec-list* in which it appears.

H404 *forall-assignment* **is** *assignment-stmt*
or *pointer-assignment-stmt*

Constraint: Any procedure referenced in a *forall-assignment*, including one referenced by a defined operation or assignment, must be pure as defined in Section 4.3.

Rationale. Pure functions are guaranteed to have no side effects, and thus have an unambiguous meaning when used in a **FORALL** statement. Experience also suggests that they form a useful class of functions for use in scientific computation, and are particularly useful when applied as data-parallel operations. For these reasons, there was a strong consensus to allow their use in **FORALL**. More general functions called from **FORALL** were also considered, but eventually rejected for lack of agreement on their desirability, ease of implementation, or the semantics of complex cases they allowed. (*End of rationale.*)

To determine the set of permitted values for each *index-name* in the *forall-header*, we introduce some simplifying notation. In the *forall-triplet-spec*, let

- $m1$ be first *subscript* ("lower bound");
- $m2$ be second *subscript* ("upper bound");
- $m3$ be the *stride*; and
- max be $\left\lfloor \frac{m2-m1+m3}{m3} \right\rfloor$.

If *stride* is missing, it is as if it were present with the value 1. *Stride* must not have the value 0. The set of permitted values is determined on entry to the statement and is $m1+(k-1)\times m3$, $k = 1, 2, ..., max$. If $max \leq 0$ for some *index-name*, the *forall-assignment* is not executed.

A **FORALL** statement assigns to memory locations specified by the *forall-assignment* for permitted values of the *index-name* variables. A program that causes multiple values to be assigned to the same location is not HPF-conforming and therefore has no defined meaning. This is a semantic constraint rather than a syntactic constraint, however; in general, it cannot be checked during compilation.

4.1.2 Interpretation of Element Array Assignments

Execution of an element array assignment consists of the following steps:

1. Evaluation in any order of the *subscript* and *stride* expressions in the *forall-triplet-spec-list*. The set of *valid combinations* of *index-name* values is then the Cartesian product of the sets defined by these triplets.

2. Evaluation of the *scalar-mask-expr* for all valid combinations of *index-name* values. The mask elements may be evaluated in any order. The set of *active combinations* of *index-name* values is the subset of the valid combinations for which the mask evaluates to **.TRUE.**

3. Evaluation in any order of the *expr* and all expressions within *variable* (in the case of *assignment-stmt*) or *target* and all expressions within *pointer-object* (in the case of *pointer-assignment-stmt.*) of the *forall-assignment* for all active combinations of *index-name* values. In the case of pointer assignment where the *target* is not a pointer, the evaluation consists of identifying the object referenced rather than computing its value.

4. Assignment of the computed *expr* values to the corresponding *variable* locations (in the case of *assignment-stmt*) or the association of the *target* values with the corresponding *pointer-object* locations (in the case of *pointer-assignment-stmt*) for all active combinations of *index-name* values. The assignments or associations may be made in any order. In the case of a pointer assignment where the *target* is not a pointer, this assignment consists of associating the *pointer-object* with the object referenced.

If the scalar mask expression is omitted, it is as if it were present with the value `.TRUE.`. The scope of an *index-name* is the `FORALL` statement itself.

A *forall-stmt* is not HPF-conforming if the result of evaluating any expression in the *forall-header* affects or is affected by the evaluation of any other expression in the *forall-header*.

Rationale. This is consistent with the handling of `DO` loop bounds and strides. Disallowing references to impure functions in a *forall-triplet-spec-list* was suggested, but the analogy to `DO` bounds was considered too strong to overlook. Note that the *scalar-mask-expr* can only invoke pure functions, which are side-effect free. Therefore, the *scalar-mask-expr* cannot affect the values of the bounds. (*End of rationale.*)

A *forall-stmt* is not HPF-conforming if it causes any atomic data object to be assigned more than one value. A data object is atomic if it contains no subobjects. For the purposes of this restriction, any assignment (including array assignment or assignment to a variable of derived type) to a non-atomic object is considered to assign to all subobjects contained by that object.

Rationale. For example, an integer variable is an atomic object, but an array of integers is an object that is not atomic. Similarly, assignment to an array section is equivalent to assignments to each individual element (which may require further reductions when the array contains objects of derived type). This restriction allows cases such as

```
FORALL ( i = 1:10 ) a(indx(i)) = b(i)
```

if and only if `indx` contains no repeated values. Note that it restricts `FORALL` behavior, but not syntax. Syntactic restrictions to enforce this behavior would be either incomplete (ie. allow undefined behavior) or exclude conceptually legal programs.

Since a function called from a *forall-asssignment* must be pure, it is impossible for that function's evaluation to affect other expressions' evaluations, either for the same combination of *index-name* values or for a different combination. In addition, it is possible that the compiler can perform more extensive optimizations because all functions are pure. (*End of rationale.*)

4.1.3 Examples of the FORALL Statement

```
FORALL (j=1:m, k=1:n) x(k,j) = y(j,k)
FORALL (k=1:n) x(k,1:m) = y(1:m,k)
```

These statements both copy columns 1 through n of array y into rows 1 through n of array x. This is equivalent to the standard Fortran 90 statement

```
x(1:n,1:m) = TRANSPOSE(y(1:m,1:n))

FORALL (i=1:n, j=1:n) x(i,j) = 1.0 / REAL(i+j-1)
```

This FORALL sets array element $x(i,j)$ to the value $\frac{1}{i+j-1}$ for values of i and j between 1 and n. In Fortran 90, the same operation can be performed by the statement

```
x(1:n,1:n) = 1.0/REAL( SPREAD((/(i,i=1,n)/),DIM=2,NCOPIES=n) &
    + SPREAD((/(j,j=1,n)/),DIM=1,NCOPIES=n) - 1 )
```

Note that the FORALL statement does not imply the creation of temporary arrays and is much more readable.

```
FORALL (i=1:n, j=1:n, y(i,j).NE.0.0) x(i,j) = 1.0 / y(i,j)
```

This statement takes the reciprocal of each nonzero element of array $y(1:n, 1:n)$ and assigns it to the corresponding element of array x. Elements of y that are zero do not have their reciprocal taken, and no assignments are made to the corresponding elements of x. This is equivalent to the standard Fortran 90 statement

```
WHERE (y(1:n,1:n) .NE. 0.0) x(1:n,1:n) = 1 / y(1:n,1:n)
```

```
TYPE monarch
  INTEGER, POINTER :: p
END TYPE monarch
TYPE(monarch) :: a(n)
INTEGER, TARGET :: b(n)

! Set up a butterfly pattern
FORALL (j=1:n)  a(j)%p => b(1+IEOR(j-1,2**k))
```

This FORALL statement sets the elements of array a to point to a permutation of the elements of b. When $n = 8$ and $k = 1$, then elements 1 through 8 of a point to elements 3, 4, 1, 2, 7, 8, 5, and 6 of b, respectively. This requires a DO loop or other control flow in Fortran 90.

```
FORALL ( i=1:n ) x(indx(i)) = x(i)
```

This FORALL statement is equivalent to the Fortran 90 array assignment

```
x(indx(1:n)) = x(1:n)
```

If *indx* contains a permutation of the integers from 1 to n, then the final contents of x will be a permutation of the original values. If *indx* contains repeated values, neither the behavior of the FORALL nor the array assignment are defined by their respective standards.

```
FORALL (i=2:4) x(i) = x(i-1) + x(i) + x(i+1)
```

If this statement is executed with

$$x = [1.0, 20.0, 300.0, 4000.0, 50000.0]$$

then after execution the new values of array x will be

$$x = [1.0, 321.0, 4320.0, 54300.0, 50000.0]$$

This has the same effect as the Fortran 90 statement

```
x(2:4) = x(1:3) + x(2:4) + x(3:5)
```

Note that it does *not* have the same effect as the Fortran 90 loop

```
DO i = 2, 4
  x(i) = x(i-1) + x(i) + x(i+1)
END DO
```

```
FORALL (i=1:n) a(i,i) = x(i)
```

This `FORALL` statement sets the elements of the main diagonal of matrix a to the elements of vector x. This cannot be done by an array assignment in Fortran 90 unless `EQUIVALENCE` or `WHERE` is also used.

```
FORALL (i=1:4) a(i,ix(i)) = x(i)
```

This `FORALL` statement sets one element in each row of matrix a to an element of vector x. The particular elements in a are chosen by the integer vector ix. If

$$x = [10.0, 20.0, 30.0, 40.0]$$

$$ix = [1, 2, 2, 4]$$

and array a represents the matrix

$$\begin{matrix} 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 1.0 & 1.0 & 1.0 & 1.0 & 1.0 \\ 2.0 & 2.0 & 2.0 & 2.0 & 2.0 \\ 3.0 & 3.0 & 3.0 & 3.0 & 3.0 \end{matrix}$$

before execution of the `FORALL`, then a will represent

$$\begin{matrix} 10.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 1.0 & 20.0 & 1.0 & 1.0 & 1.0 \\ 2.0 & 30.0 & 2.0 & 2.0 & 2.0 \\ 3.0 & 3.0 & 3.0 & 3.0 & 40.0 \end{matrix}$$

after its execution. This operation cannot be accomplished with a single array assignment in Fortran 90.

```
FORALL (k=1:9) x(k) = SUM(x(1:10:k))
```

This `FORALL` statement computes nine sums of subarrays of `x`. (`SUM` is allowed in a `FORALL` because Fortran 90 intrinsic functions are pure; see Section 4.3.) If before the `FORALL`

$$x = [1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0]$$

then after the `FORALL`

$$x = [55.0, 25.0, 22.0, 15.0, 7.0, 8.0, 9.0, 10.0, 11.0, 10.0]$$

This computation cannot be done by Fortran 90 array expressions alone.

4.1.4 Scalarization of the FORALL Statement

One way to understand the semantics of the **FORALL** statement is to exhibit a naive translation to scalar Fortran 90 code. We provide such a translation below.

> *Advice to implementors.* Note, however, that such a translation is meant for illustration rather than as the definitive reference to the **FORALL** semantics of or practical implementation in the compiler. In particular, implementing a **FORALL** using **DO** loops imposes an apparent order on the operations that is not implied by the formal definition. Additionally, compiler analysis of particular cases may allow significant simplification and optimization. For example, if the array assigned in a **FORALL** statement is not referenced in any other expression in the **FORALL** (including its use in functions called from the **FORALL**), it is legal and, on many machines, more efficient to perform the computations and final assignments in a single loop nest. Also note the discussion at the end of this section regarding other difficulties of a Fortran 90 translation. (*End of advice to implementors.*)

A *forall-stmt* of the form

FORALL $(v_1{=}l_1 : u_1 : s_1, v_2{=}l_1 : u_2 : s_2, \ldots, v_n{=}l_n : u_n : s_n, mask)$ $a(e_1, \ldots, e_m){=}rhs$

is equivalent to the following code:

```
! Evaluate subscript and stride expressions.
! These assignments may be executed in any order.
templ1 = l1
tempu1 = u1
temps1 = s1
templ2 = l2
tempu2 = u2
temps2 = s2
  ...
templn = ln
tempun = un
tempsn = sn

! Evaluate the scalar mask expression, and evaluate the
! forall-assignment subexpressions where the mask is true.
! The iterations of this loop nest may be executed in any order.
! The assignments in the loop body may be executed in any order,
! provided that the mask element is evaluated before any other
! expression in the same iteration.
! The loop body need not be executed atomically.
! The DO statements may be nested in any order
DO v1=templ1,tempu1,temps1
  DO v2=templ2,tempu2,temps2
    ...
      DO vn=templn,tempun,tempsn
        tempmask(v1,v2,...,vn) = mask
```

```
        IF (tempmask(v1,v2,...,vn)) THEN
          temprhs(v1,v2,...,vn) = rhs
          tempe1(v1,v2,...,vn) = e1
          tempe2(v1,v2,...,vn) = e2
          ....
          tempem(v1,v2,...,vn) = em
        END IF
      END DO
    ...
  END DO
END DO

! Perform the assignment of these values to the corresponding
! elements of the array on the left-hand side.
! The iterations of this loop nest may be executed in any order.
! The DO statements may be nested in any order.
DO v1=templ1,tempu1,temps1
  DO v2=templ2,tempu2,temps2
    ...
      DO vn=templn,tempun,tempsn
        IF (tempmask(v1,v2,...,vn)) THEN
          a(tempe1(v1,v2,...,vn),...,tempem(v1,v2,...,vn)) =     &
            temprhs(v1,v2,...,vn)
        END IF
      END DO
    ...
  END DO
END DO
```

The scalarization of a **FORALL** statement containing a pointer assignment is similar, replacing the assignments to *temprhs* and a with pointer assignments.

Advice to implementors. Several subtleties are not specified in the above outline to promote readability. When *rhs* is an array-valued expression, then several of the statements cannot be translated directly into Fortran 90. In particular, at least one of the e_i will be a triplet; both bounds and stride must be saved in $tempe_i$, possibly by using derived type assignment or adding a dimension to the data structure. The translation of the subscripts in the final assignment to a must also be generalized to handle triplets. Storage allocation for *temprhs* may be complicated by the fact that it must store arrays (possibly with different sizes for different values of $v_1, \ldots, v_n$). If the *forall-assignment* is a *pointer-assignment-stmt*, then a suitable derived type must be produced for *temprhs*. The assignments to $tempe_1, \ldots, tempe_m$ must, however, remain true (integer) assignments. Finally, there may also be more than seven indexes; this may forbid a direct translation on implementations that support a limited number of dimensions in arrays. (*End of advice to implementors.*)

4.1.5 Consequences of the Definition of the FORALL Statement

Rationale. The *scalar-mask-expr* may depend on the *index-name* values. This allows a wide range of masking operations.

A syntactic consequence of the semantic rule that no two execution instances of the body may assign to the same atomic data object is that each of the *index-name* variables must appear on the left-hand side of a *forall-assignment.* The converse is not true (i.e., using all *index-name* variables on the left-hand side does not guarantee there will be no interference). Because the condition is not sufficient, it does not appear a syntax constraint. This also allows for easier future extensions for private variables or other syntactic sugar.

Right-hand sides and expressions on the left hand side of a *forall-assignment* are defined as evaluated only for combinations of *index-names* for which the *scalar-mask-expr* evaluates to `.TRUE.`. This has implications when the masked computation might create an error condition. For example,

```
FORALL (i=1:n, y(i).NE.0.0) x(i) = 1.0 / y(i)
```

does not cause a division by zero. (*End of rationale.*)

4.2 The FORALL Construct

The `FORALL` construct is a generalization of the `FORALL` statement allowing multiple assignments, masked array assignments, and nested `FORALL` statements and constructs to be controlled by a single *forall-triplet-spec-list.*

4.2.1 General Form of the FORALL Construct

Rule R215 of the Fortran 90 standard for *executable-construct* is extended to include the *forall-construct.*

H405 *forall-construct* **is** `FORALL` *forall-header*
forall-body-stmt
[*forall-body-stmt*] ...
`END FORALL`

H406 *forall-body-stmt* **is** *forall-assignment*
or *where-stmt*
or *where-construct*
or *forall-stmt*
or *forall-construct*

Constraint: Any procedure referenced in a *forall-body-stmt*, including one referenced by a defined operation or assignment, must be pure as defined in Section 4.3.

Constraint: If a *forall-stmt* or *forall-construct* is nested in a *forall-construct*, then the inner `FORALL` may not redefine any *index-name* used in the outer *forall-construct.*

Rationale. These statements are allowed in a **FORALL** construct because they are defined as forms of assignment in Fortran 90 and HPF. The intent is that *forall-construct*, like *forall-stmt*, is a block assignment rather than a general-purpose "parallel loop." (*End of rationale.*)

To determine the set of permitted values for an *index-name*, we introduce some simplifying notation. In the *forall-triplet-spec*, let

- $m1$ be the first *subscript* ("lower bound");
- $m2$ be the second *subscript* ("upper bound");
- $m3$ be the *stride*; and
- max be $\left\lfloor \frac{m2-m1+m3}{m3} \right\rfloor$.

If *stride* is missing, it is as if it were present with the value 1. The set of permitted values is determined on entry to the construct and is $m1 + (k-1) \times m3$, $k = 1, 2, ..., max$. The expression *stride* must not have the value 0. If for some *index-name* $max \leq 0$, no *forall-body-stmt* is executed.

Each assignment nested within a **FORALL** construct assigns to memory locations specified by the *forall-assignment* for permitted values of the *index-name* variables. A program that causes multiple values to be assigned to the same location by a single statement is not HPF-conforming and therefore has no defined meaning. An HPF-conforming program may, however, assign to the same location in syntactically different assignment statements. This is a semantic constraint rather than a syntactic constraint, however; in general, it cannot be checked during compilation.

4.2.2 Interpretation of the FORALL Construct

Execution of a **FORALL** construct consists of the following steps:

1. Evaluation in any order of the *subscript* and *stride* expressions in the *forall-triplet-spec-list*. The set of *valid combinations* of *index-name* values is then the Cartesian product of the sets defined by these triplets.

2. Evaluation of the *scalar-mask-expr* for all valid combinations of *index-name* values. The mask elements may be evaluated in any order. The set of *active combinations* of *index-name* values is the subset of the valid combinations for which the mask evaluates to **.TRUE.**

3. Execute the *forall-body-stmts* in the order they appear. Each statement is executed completely (that is, for all active combinations of *index-name* values) according to the following interpretation:

   (a) Statements in the *forall-assignment* category (i.e. assignment statements and pointer assignment statements) evaluate the *expr* and all expressions within *variable* (in the case of *assignment-stmt*) or *target* and all expressions within *pointer-object* (in the case of *pointer-assignment-stmt*) of the *forall-assignment* for all active combinations of *index-name* values. These evaluations may be done

in any order. The *expr* values are then assigned to the corresponding *variable* locations (in the case of *assignment-stmt*) or the *target* values are associated with the corresponding *pointer-object* locations (in the case of *pointer-assignment-stmt*). The assignment or association operations may also be performed in any order.

(b) Statements in the *where-stmt* and *where-construct* categories evaluate their *mask-expr* for all active combinations of values of *index-names*. All elements of all masks may be evaluated in any order. The **WHERE** statement's assignment (or assignments within the **WHERE** branch of the construct) are then executed in order using the above interpretation of array assignments within the **FORALL**, but the only array elements assigned are those selected by both the active *index-name* values and the **WHERE** mask. Finally, the assignments in the **ELSEWHERE** branch are executed if that branch is present. The assignments here are also treated as array assignments, but elements are only assigned if they are selected by both the active combinations and by the negation of the **WHERE** mask.

(c) Statements in the *forall-stmt* and *forall-construct* categories first evaluate the *subscript* and *stride* expressions in the *forall-triplet-spec-list* for all active combinations of the outer **FORALL** constructs. The set of valid combinations of *index-names* for the inner **FORALL** is then the union of the sets defined by these bounds and strides for each active combination of the outer *index-names*, the outer *index names* being included in the combinations generated for the inner **FORALL**. The scalar mask expression is then evaluated for all valid combinations of the inner **FORALL**'s *index-names* to produce the set of active combinations. If there is no scalar mask expression, it is as if it were present with the constant value **.TRUE.** Each statement in the inner **FORALL** is then executed for each active combination (of the inner **FORALL**), recursively following the interpretations given in this section.

If the scalar mask expression is omitted, it is as if it were present with the value **.TRUE.**

The scope of an *index-name* is the **FORALL** construct itself.

Each *forall-assignment* must obey the same restrictions in a *forall-construct* as in a simple *forall-stmt.* In addition, each *where-stmt* or assignment nested within a *where-construct* must obey these restrictions. (Note that any innermost statement within nested **FORALL** constructs must fall into one of these two categories.) For example, an assignment may not cause the same array element to be assigned more than once. Different statements may, however, assign to the same array element, and assignments made in one statement may affect the execution of a later statement.

4.2.3 Examples of the FORALL Construct

```
FORALL ( i=2:n-1, j=2:n-1 )
  a(i,j) = a(i,j-1) + a(i,j+1) + a(i-1,j) + a(i+1,j)
  b(i,j) = a(i,j)
END FORALL
```

This **FORALL** is equivalent to the two Fortran 90 statements

```
a(2:n-1,2:n-1) = a(2:n-1,1:n-2)+a(2:n-1,3:n)        &
                          +a(1:n-2,2:n-1)+a(3:n,2:n-1)
b(2:n-1,2:n-1) = a(2:n-1,2:n-1)
```

In particular, note that the assignment to array b uses the values of array a computed in the first statement, not the values before the **FORALL** began execution.

```
FORALL ( i=1:n-1 )
  FORALL ( j=i+1:n )
    a(i,j) = a(j,i)
  END FORALL
END FORALL
```

This **FORALL** construct assigns the transpose of the lower triangle of array a (i.e., the section below the main diagonal) to the upper triangle of a. For example, if $n = 5$ and a originally contained the matrix

$$\begin{array}{rrrrr} 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 1.0 & 1.0 & 1.0 & 1.0 & 1.0 \\ 2.0 & 4.0 & 8.0 & 16.0 & 32.0 \\ 3.0 & 9.0 & 27.0 & 81.0 & 243.0 \\ 4.0 & 16.0 & 64.0 & 256.0 & 1024.0 \end{array}$$

then after the **FORALL** it would contain

$$\begin{array}{rrrrr} 0.0 & 1.0 & 2.0 & 3.0 & 4.0 \\ 1.0 & 1.0 & 4.0 & 9.0 & 16.0 \\ 2.0 & 4.0 & 8.0 & 27.0 & 64.0 \\ 3.0 & 9.0 & 27.0 & 81.0 & 256.0 \\ 4.0 & 16.0 & 64.0 & 256.0 & 1024.0 \end{array}$$

This cannot be done using array expressions without introducing mask expressions.

```
FORALL ( i=1:5 )
  WHERE ( a(i,:) .NE. 0.0 )
    a(i,:) = a(i-1,:) + a(i+1,:)
  ELSEWHERE
    b(i,:) = a(6-i,:)
  END WHERE
END FORALL
```

This **FORALL** construct, when executed with the input arrays

$$a = \begin{pmatrix} 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 1.0 & 1.0 & 1.0 & 0.0 & 1.0 \\ 2.0 & 2.0 & 0.0 & 2.0 & 2.0 \\ 3.0 & 0.0 & 3.0 & 3.0 & 3.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \end{pmatrix}, \quad b = \begin{pmatrix} 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 10.0 & 10.0 & 10.0 & 10.0 & 10.0 \\ 20.0 & 20.0 & 20.0 & 20.0 & 20.0 \\ 30.0 & 30.0 & 30.0 & 30.0 & 30.0 \\ 40.0 & 40.0 & 40.0 & 40.0 & 40.0 \end{pmatrix}$$

will produce as results

$$a = \begin{pmatrix} 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 2.0 & 2.0 & 0.0 & 0.0 & 2.0 \\ 4.0 & 1.0 & 0.0 & 3.0 & 4.0 \\ 2.0 & 0.0 & 0.0 & 2.0 & 2.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \end{pmatrix}, \quad b = \begin{pmatrix} 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 10.0 & 10.0 & 10.0 & 2.0 & 10.0 \\ 20.0 & 20.0 & 0.0 & 20.0 & 20.0 \\ 30.0 & 2.0 & 30.0 & 30.0 & 30.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \end{pmatrix}$$

Note that, as with **WHERE** statements in ordinary Fortran 90, assignments in the **WHERE** branch may affect computations in the **ELSEWHERE** branch.

4.2.4 Scalarization of the FORALL Construct

Advice to implementors. As with the **FORALL** statement, the following translations of **FORALL** constructs to **DO** loops are meant to illustrate the meaning, not necessarily to serve as an implementation guide. The caveats for the **FORALL** statement scalarization apply here as well. (*End of advice to implementors.*)

A *forall-construct* of the form:

FORALL (... e_1 ... e_2 ... e_n ...)
  s_1
  s_2
  ...
  s_n
END FORALL

where each s_i is a *forall-assignment* is equivalent to the following code:

$temp_1$ = e_1
$temp_2$ = e_2
...
$temp_n$ = e_n
FORALL (... $temp_1$ ... $temp_2$ ... $temp_n$ ...) s_1
FORALL (... $temp_1$ ... $temp_2$ ... $temp_n$ ...) s_2
  ...
FORALL (... $temp_1$ ... $temp_2$ ... $temp_n$ ...) s_n

When the s_i are **FORALL** or **WHERE** statements or constructs, then the **FORALL** statements above must be replaced with **FORALL** constructs (since **FORALL** statements can only contain assignments). The scalarizations below must then be applied to the shortened **FORALL** constructs.

A *forall-construct* of the form:

FORALL (v_1=l_1 : u_1 : s_1 , $mask_1$)
  WHERE ($mask_2$)
    $a(l_2 : u_2 : s_2)$ = rhs_1
  ELSEWHERE
    $a(l_3 : u_3 : s_3)$ = rhs_2
  END WHERE
END FORALL

is equivalent to the following code:

```
! Evaluate subscript and stride expressions.
! These assignments can be made in any order.
templ1 = l1
tempu1 = u1
temps1 = s1

! Evaluate the FORALL mask expression.
! The iterations of this loop may be executed in any order.
DO v1=templ1,tempu1,temps1
 tempmask1(v1) = mask1
END DO

! Evaluate the bounds and masks for the WHERE.
! The iterations of this loop may be executed in any order.
! The loop body need not be executed atomically.
DO v1=templ1,tempu1,temps1
  IF (tempmask1(v1)) THEN
    tempmask2(v1) = mask2
  END IF
END DO

! Evaluate the WHERE branch.
! The iterations of this loop may be executed in any order.
! The assignments in the loop body may be executed in any order.
! The loop body need not be executed atomically.
DO v1=templ1,tempu1,temps1
  IF (tempmask1(v1)) THEN
    tmpl2(v1) = l2
    tmpu2(v1) = u2
    tmps2(v1) = s2
    WHERE ( tempmask2(v1) )
      temprhs1(v1) = rhs1
    END WHERE
  END IF
END DO
! The iterations of this loop may be executed in any order.
! The loop body need not be executed atomically.
DO v1=templ1,tempu1,temps1
  IF (tempmask1(v1)) THEN
    WHERE ( tempmask2(v1) )
      a(tmpl2(v1):tmpu2(v1):tmps2(v1)) = temprhs1(v1)
    END WHERE
  END IF
END DO

! Evaluate the ELSEWHERE branch.
```

```
! The iterations of this loop may be executed in any order.
! The assignments in the loop body may be executed in any order.
! The loop body need not be executed atomically.
DO v1=templ1,tempu1,temps1
  IF (tempmask1(v1)) THEN
    tmpl3(v1) = l3
    tmpu3(v1) = u3
    tmps3(v1) = s3
    WHERE ( .NOT. tempmask2(v1) )
      temprhs2(v1) = rhs2
    END WHERE
  END IF
END DO
! The iterations of this loop may be executed in any order.
! The loop body need not be executed atomically.
DO v1=templ1,tempu1,temps1
  IF (tempmask1(v1)) THEN
    WHERE ( .NOT. tempmask2(v1) )
      a(tmpl3(v1):tmpu3(v1):tmps3(v1)) = temprhs2(v1)
    END WHERE
  END IF
END DO
```

Advice to implementors. Note that the assignments to $tempmask_2$ and $temprhs_i$ are array assignments and require special treatment (including saving of shape information) similar to that for array assignments in the FORALL statement scalarization. The extension to multiple dimensions (in either the FORALL index space or the array dimensions) is straightforward. If there are multiple statements in a branch of the WHERE construct, each statement will generate two loops similar to those shown above. (*End of advice to implementors.*)

A *forall-construct* of the form:

```
FORALL ( v1=l1:u1:s1, mask1 )
  FORALL ( v2=l2:u2:s2, mask2 )
    a(e1) = rhs1
    b(e2) = rhs2
  END FORALL
END FORALL
```

is equivalent to the following Fortran 90 code:

```
! Evaluate subscript and stride expressions and outer mask.
! These assignments may be executed in any order.
templ1 = l1
tempu1 = u1
temps1 = s1
! The iterations of this loop may be executed in any order.
DO v1=templ1,tempu1,temps1
```

```
  tempmask1(v1) = mask1                                                    1
END DO                                                                     2
                                                                           3
! Evaluate the inner FORALL bounds, etc                                    4
! The iterations of this loop may be executed in any order.               5
! The assignments in the loop body may be executed in any order,          6
! provided that the mask bounds are computed before the mask itself.      7
! The loop body need not be executed atomically.                          8
DO v1=templ1,tempu1,temps1                                                 9
  IF (tempmask1(v1)) THEN                                                  10
    templ2(v1) = l2                                                        11
    tempu2(v1) = u2                                                        12
    temps2(v1) = s2                                                        13
    DO v2 = templ2(v1),tempu2(v1),temps2(v1)                               14
      tempmask2(v1,v2) = mask2                                             15
    END DO                                                                 16
  END IF                                                                   17
END DO                                                                     18
                                                                           19
! Evaluate first statement                                                 20
! The iterations of this loop may be executed in any order.               21
! The assignments in this loop body may be executed in any order.         22
! The loop body need not be executed atomically.                          23
DO v1=templ1,tempu1,temps1                                                 24
  IF (tempmask1(v1)) THEN                                                  25
    DO v2 = templ2(v1),tempu2(v1),temps2(v1)                               26
      IF ( tempmask2(v1,v2) ) THEN                                         27
        temprhs1(v1,v2) = rhs1                                             28
        tmpe1(v1,v2) = e1                                                  29
      END IF                                                               30
    END DO                                                                 31
  END IF                                                                   32
END DO                                                                     33
! The iterations of this loop may be executed in any order.               34
DO v1=templ1,tempu1,temps1                                                 35
  IF (tempmask1(v1)) THEN                                                  36
    DO v2 = templ2(v1),tempu2(v1),temps2(v1)                               37
      IF ( tempmask2(v1,v2) ) THEN                                         38
        a(tmpe1(v1,v2)) = temprhs1(v1,v2)                                  39
      END IF                                                               40
    END DO                                                                 41
  END IF                                                                   42
END DO                                                                     43
                                                                           44
! Evaluate second statement.                                               45
! Ordering constraints are as for the first statement.                    46
DO v1=templ1,tempu1,temps1                                                 47
  IF (tempmask1(v1)) THEN                                                  48
```

```
      DO v2 = templ2(v1),tempu2(v1),temps2(v1)
        IF ( tempmask2(v1,v2) ) THEN
          temprhs2(v1,v2) = rhs2
          tmpe2(v1,v2) = e2
        END IF
      END DO
    END IF
  END DO
  DO v1=templ1,tempu1,temps1
    IF (tempmask1(v1)) THEN
      DO v2 = templ2(v1),tempu2(v1),temps2(v1)
        IF ( tempmask2(v1,v2) ) THEN
          b(tmpe2(v1,v2)) = temprhs2(v1,v2)
        END IF
      END DO
    END IF
  END DO
```

Again, the extensions to higher dimensions are straightforward, as is the extension to deeper nesting levels.

Advice to implementors. Note that each statement at the deepest nesting level will generate two loops of the types shown. (*End of advice to implementors.*)

4.2.5 Consequences of the Definition of the FORALL Construct

Rationale.

A block **FORALL** means roughly the same thing as does replicating the **FORALL** header in front of each array assignment statement in the block, except that any expressions in the **FORALL** header are evaluated only once, rather than being re-evaluated before each of the statements in the body. The exceptions to this rule are nested **FORALL** statements and **WHERE** statements, which introduce syntactic and functional complications into the copying.

One may think of a block **FORALL** as synchronizing twice per contained assignment statement: once after handling the right-hand side and other expressions but before performing assignments, and once after all assignments have been performed but before commencing the next statement. In practice, appropriate analysis will often permit the compiler to eliminate unnecessary synchronizations.

In general, any expression in a **FORALL** is evaluated only for valid combinations of all surrounding *index-names* for which all the scalar mask expressions are **.TRUE.**

Nested **FORALL** bounds and strides can depend on outer **FORALL** *index-names*. They cannot redefine those names, even temporarily (if they did, there would be no way to avoid multiple assignments to the same array element).

Statements can use the results of computations in lexically earlier statements, including computations done for other name values. However, an assignment never uses a value assigned in the same statement by another *index-name* value combination.

(*End of rationale.*)

4.3 Pure Procedures

A *pure function* is one that obeys certain syntactic constraints that ensure it produces no side effects. This means that the only effect of a pure function reference on the state of a program is to return a result—it does not modify the values, pointer associations, or data mapping of any of its arguments or global data, and performs no external I/O. A *pure subroutine* is one that produces no side effects except for modifying the values and/or pointer associations of `INTENT(OUT)` and `INTENT(INOUT)` arguments. These properties are declared by a new attribute (the `PURE` attribute) of the the procedure.

A pure procedure (i.e., function or subroutine) may be used in any way that a normal procedure can. However, a procedure is required to be pure if it is used in any of the following contexts:

- The mask or body of a `FORALL` statement or construct;
- Within the body of a pure procedure; or
- As an actual argument in a pure procedure reference.

Rationale.

The freedom from side effects of a pure function allows the function to be invoked concurrently in a `FORALL` without such undesirable consequences as nondeterminism, and additionally assists the efficient implementation of concurrent execution. Syntactic constraints (rather than semantic constraints on behavior) are used to enable compiler checking.

The HPF Journal of Development also proposes allowing elemental invocation of pure procedures with scalar arguments.

(*End of rationale.*)

4.3.1 Pure Procedure Declaration and Interface

If a user-defined procedure is used in a context that requires it to be pure, then its interface must be explicit in the scope of that use, and that interface must specify the `PURE` attribute. This attribute is specified in the *function-stmt* or *subroutine-stmt* by an extension of rules R1217 (for *prefix*) and R1220 (for *subroutine-stmt*) in the Fortran 90 standard. Rule R1216 (for *function-stmt*) is not changed, but is rewritten here as Rule H409 for clarity.

H407 *prefix* **is** *prefix-spec* [*prefix-spec*] ...

H408 *prefix-spec* **is** *type-spec*
or `RECURSIVE`
or `PURE`
or *extrinsic-prefix*

H409 *function-stmt* **is** [*prefix*] `FUNCTION` *function-name function-stuff*

H410 *function-stuff* **is** ([*dummy-arg-name-list*]) [`RESULT` (*result-name*)]

H411 *subroutine-stmt* **is** [*prefix*] `SUBROUTINE` *subroutine-name subroutine-stuff*

H412 *subroutine-stuff* **is** [([*dummy-arg-list*])]

Constraint: A *prefix* must contain at most one of each variety of *prefix-spec.*

Constraint: The *prefix* of a *subroutine-stmt* must not contain a *type-spec.*

(For a discussion of the *extrinsic-prefix* (Rule H601), see Section 6.2.)

Intrinsic functions, including the HPF intrinsic functions, are always pure and require no explicit declaration of this fact. Intrinsic subroutines are pure if they are elemental (i.e., `MVBITS`) but not otherwise. Functions in the HPF library are declared to be pure. A statement function is pure if and only if all functions that it references are pure.

A procedure with the `PURE` attribute is referred to as a "pure procedure" in the following constraints.

4.3.1.1 Pure function definition

The following constraints are added to Rule R1215 in Section 12.5.2.2 of the Fortran 90 standard (defining *function-subprogram*):

Constraint: The *specification-part* of a pure function must specify that all dummy arguments have `INTENT(IN)` except procedure arguments and arguments with the `POINTER` attribute.

Constraint: A local variable declared in the *specification-part* or *internal-subprogram-part* of a pure function must not have the `SAVE` attribute.

> *Advice to users.* Note local variable initialization in a *type-declaration-stmt* or a *data-stmt* implies the `SAVE` attribute; therefore, such initialization is also disallowed. (*End of advice to users.*)

Constraint: The *execution-part* and *internal-subprogram-part* of a pure function may not use a dummy argument, a global variable, or an object that is storage associated with a global variable, or a subobject thereof, in the following contexts:

- As the assignment variable of an *assignment-stmt*;
- As a `DO` variable or implied `DO` variable, or as an *index-name* in a *forall-triplet-spec*;
- As an *input-item* in a *read-stmt*;
- As an *internal-file-unit* in a *write-stmt*;
- As an `IOSTAT=` or `SIZE=` specifier in an I/O statement.
- In an *assign-stmt*;
- As the *pointer-object* or *target* of a *pointer-assignment-stmt*;
- As the *expr* of an *assignment-stmt* whose assignment variable is of a derived type, or is a pointer to a derived type, that has a pointer component at any level of component selection;
- As an *allocate-object* or *stat-variable* in an *allocate-stmt* or *deallocate-stmt*, or as a *pointer-object* in a *nullify-stmt*; or
- As an actual argument associated with a dummy argument with `INTENT (OUT)` or `INTENT(INOUT)` or with the `POINTER` attribute.

Constraint: Any procedure referenced in a pure function, including one referenced via a defined operation or assignment, must be pure.

Constraint: A dummy argument or the dummy result of a pure function may be explicitly aligned only with another dummy argument or the dummy result, and may not be explicitly distributed or given the `INHERIT` attribute.

Constraint: In a pure function, a local variable may be explicitly aligned only with another local variable, a dummy argument, or the result variable. A local variable may not be explicitly distributed.

Constraint: In a pure function, a dummy argument, local variable, or the result variable must not have the `DYNAMIC` attribute.

Constraint: In a pure function, a global variable must not appear in a *realign-directive* or *redistribute-directive*.

Constraint: A pure function must not contain a *print-stmt*, *open-stmt*, *close-stmt*, *backspace-stmt*, *endfile-stmt*, *rewind-stmt*, *inquire-stmt*, or a *read-stmt* or *write-stmt* whose *io-unit* is an *external-file-unit* or *.

Constraint: A pure function must not contain a *pause-stmt* or *stop-stmt*.

The above constraints are designed to guarantee that a pure function is free from side effects (i.e., modifications of data visible outside the function), which means that it is safe to reference concurrently, as explained earlier.

Rationale.

It is worth mentioning why the above constraints are sufficient to eliminate side effects.

The first constraint (requiring explicit `INTENT(IN)`) declares behavior that is ensured by the following rules. It is not technically necessary, but is included for consistency with the explicit declaration rules for defined operators. Note that `POINTER` arguments may not have the `INTENT` attribute; the restrictions below ensure that `POINTER` arguments also behave as if they had `INTENT(IN)`, for both the argument itself and the object pointed to.

The second constraint (disallowing `SAVE` variables) ensures that a pure function does not retain an internal state between calls, which would allow side-effects between calls to the same procedure.

The third constraint (the restrictions on use of global variables and dummy arguments) ensures that dummy arguments and global variables are not modified by the function. In the case of a dummy or global pointer, this applies to both its pointer association and its target value, so it cannot be subject to a pointer assignment or to an `ALLOCATE`, `DEALLOCATE`, or `NULLIFY` statement. Incidentally, these constraints imply that only local variables and the dummy result variable can be subject to assignment or pointer assignment.

In addition, a dummy or global data object cannot be the *target* of a pointer assignment (i.e., it cannot be used as the right hand side of a pointer assignment to a local pointer or to the result variable), for then its value could be modified via the pointer. (An alternative approach would be to allow such objects to be pointer targets, but

disallow assignments to those pointers; syntactic constraints to allow this would be even more draconian than these.)

In connection with the last point, it should be noted that an ordinary (as opposed to pointer) assignment to a variable of derived type that has a pointer component at any level of component selection may result in a *pointer* assignment to the pointer component of the variable. That is certainly the case for an intrinsic assignment. In that case, the expression on the right hand side of the assignment has the same type as the assignment variable, and the assignment results in a pointer assignment of the pointer components of the expression result to the corresponding components of the variable (see section 7.5.1.5 of the Fortran 90 standard). However, it may also be the case for a *defined* assignment to such a variable, even if the data type of the expression has no pointer components; the defined assignment may still involve pointer assignment of part or all of the expression result to the pointer components of the assignment variable. Therefore, a dummy or global object cannot be used as the right hand side of any assignment to a variable of derived type with pointer components, for then it, or part of it, might be the target of a pointer assignment, in violation of the restriction mentioned above.

(Incidentally, the last two paragraphs only prevent the reference of a dummy or global object as the *only* object on the right hand side of a pointer assignment or an assignment to a variable with pointer components. There are no constraints on its reference as an operand, actual argument, subscript expression, etc. in these circumstances.)

Finally, a dummy or global data object cannot be used in a procedure reference as an actual argument associated with a dummy argument of `INTENT(OUT)` or `INTENT(INOUT)` or with a dummy pointer, for then it may be modified by the procedure reference. This constraint, like the others, can be statically checked, since any procedure referenced within a pure function must be either a pure function, which does not modify its arguments, or a pure subroutine, whose interface must specify the `INTENT` or `POINTER` attributes of its arguments (see below). Incidentally, notice that in this context it is assumed that an actual argument associated with a dummy pointer is modified, since Fortran 90 does not allow its intent to be specified.

The fourth constraint (only pure procedures may be called) ensures that all procedures called from a pure function are themselves side-effect free, except, in the case of subroutines, for modifying actual arguments associated with dummy pointers or dummy arguments with `INTENT(OUT)` or `INTENT(INOUT)`. As we have just explained, it can be checked that global or dummy objects are not used in such arguments, which would violate the required side-effect freedom.

Constraints 5 and 6 restrict the explicit declaration of the mapping of local variables and the dummy arguments and dummy results. This is because the function may be invoked concurrently, with each invocation active on a subset of processors specific to that invocation, and operating on data that are mapped to that processor subset. Indeed, in an optimising implementation, the caller may well automatically arrange the mapping of the actual arguments and result according to the context, e.g. to maximise concurrency in a `FORALL`, and/or to reduce communication, taking into account the mappings of other arguments, other terms in the expression, the assignment variable, etc. Thus, a dummy argument or result may not appear in a mapping directive that fixes its location with respect to the processor array (e.g. it may not be aligned with a

global variable or template, or be explicitly distributed, or given the inherit attribute, all of which would remove the caller's freedom to determine the actual's mapping as described above). The only type of mapping information that may be specified for the dummy arguments and result is their alignment with each other; this will provide useful information to the caller about their required *relative* mappings. For similar reasons, local variables may be aligned with the dummy arguments or result (either directly or through other local variables), but may not have arbitrary mappings.

Constraints 7 and 8 prevent any realignment and redistribution of data within a pure function (another type of side effect).

The penultimate constraint prevents external I/O and file operations, whose order would be non-deterministic in the context of concurrent execution. Note that internal I/O *is* allowed, provided that it does not modify global variables or dummy arguments.

Finally, the last constraint disallows `PAUSE` and `STOP` statements. A `PAUSE` statement requires input and so is disallowed for the same reason as I/O. A `STOP` brings execution to a halt, which is a rather drastic side effect.

(*End of rationale.*)

4.3.1.2 Pure subroutine definition

The following constraints are added to Rule R1219 in Section 12.5.2.3 of the Fortran 90 standard (defining *subroutine-subprogram*):

Constraint: The *specification-part* of a pure subroutine must specify the intents of all dummy arguments except procedure arguments and arguments that have the `POINTER` attribute.

Constraint: A local variable declared in the *specification-part* or *internal-function-part* of a pure subroutine must not have the `SAVE` attribute.

Constraint: The *execution-part* or *internal-subprogram-part* of a pure subroutine must not use a dummy parameter with `INTENT(IN)`, a global variable, or an object that is storage associated with a global variable, or a subobject thereof, in the following contexts:

- As the assignment variable of an *assignment-stmt*;
- As a `DO` variable or implied `DO` variable, or as a *index-name* in a *forall-triplet-spec*;
- As an *input-item* in a *read-stmt*;
- As an *internal-file-unit* in a *write-stmt*;
- As an `IOSTAT=` or `SIZE=` specifier in an I/O statement.
- In an *assign-stmt*;
- As the *pointer-object* or *target* of a *pointer-assignment-stmt*;
- As the *expr* of an *assignment-stmt* whose assignment variable is of a derived type, or is a pointer to a derived type, that has a pointer component at any level of component selection;

- As an *allocate-object* or *stat-variable* in an *allocate-stmt* or *deallocate-stmt*, or as a *pointer-object* in a *nullify-stmt*;
- As an actual argument associated with a dummy argument with `INTENT (OUT)` or `INTENT(INOUT)` or with the `POINTER` attribute.

Constraint: Any procedure referenced in a pure subroutine, including one referenced via a defined operation or assignment, must be pure.

Constraint: A dummy argument of a pure subroutine may be explicitly aligned only with another dummy argument, and may not be explicitly distributed or given the `INHERIT` attribute.

Constraint: In a pure subroutine, a local variable may be explicitly aligned only with another local variable or a dummy argument. A local variable may not be explicitly distributed.

Constraint: In a pure subroutine, a dummy argument or local variable must not have the `DYNAMIC` attribute.

Constraint: In a pure subroutine, a global variable must not appear in a *realign-directive* or *redistribute-directive*.

Constraint: A pure subroutine must not contain a *print-stmt*, *open-stmt*, *close-stmt*, *backspace-stmt*, *endfile-stmt*, *rewind-stmt*, *inquire-stmt*, or a *read-stmt* or *write-stmt* whose *io-unit* is an *external-file-unit* or *.

Constraint: A pure subroutine must not contain a *pause-stmt* or *stop-stmt.*

Rationale.

The constraints for pure subroutines are based on the same principles as for pure functions, except that side effects to `INTENT(OUT)` and `INTENT(INOUT)` dummy arguments are permitted. Pointer dummy arguments are always treated as `INTENT(INOUT)`.

Pure subroutines are included to allow subroutine calls from pure procedures in a safe way, and to allow *forall-assignments* to be defined assignments.

(*End of rationale.*)

4.3.1.3 Pure procedure interfaces

To define interface specifications for pure procedures, the following constraints are added to Rule R1204 in Section 12.3.2.1 of the Fortran 90 standard (defining *interface-body*):

Constraint: An *interface-body* of a pure procedure must specify the intents of all dummy arguments except `POINTER` and procedure arguments.

The procedure characteristics defined by an interface body must be consistent with the procedure's definition. Regarding pure procedures, this is interpreted as follows:

- A procedure that is declared pure at its definition may be declared pure in an interface body, but this is not required.

- A procedure that is not declared pure at its definition must not be declared pure in an interface body.

That is, if an interface body contains a **PURE** attribute, then the corresponding procedure definition must also contain it, though the reverse is not true. When a procedure definition with a **PURE** attribute is compiled, the compiler may check that it satisfies the necessary constraints.

4.3.2 Pure Procedure Reference

To define pure procedure references, the following extra constraint is added to Rules R1209 and R1210 in Section 12.4.1 of the Fortran 90 standard (defining *function-reference* and *call-stmt*):

Constraint: In a reference to a pure procedure, a *procedure-name actual-arg* must be the name of a pure procedure.

Rationale. This constraint ensures that the purity of a procedure cannot be undermined by allowing it to call a non-pure procedure. (*End of rationale.*)

4.3.3 Examples of Pure Procedure Usage

Pure functions may be used in expressions in **FORALL** statements and constructs, unlike general functions. Several examples of this are given below.

```
! This statement function is pure since it does not reference
! any other functions
REAL myexp
myexp(x) = 1 + x + x*x/2.0 + x*x*x/6.0
FORALL ( i = 1:n ) a(i) = myexp( a(i+1) )
...
! Intrinsic functions are always pure
FORALL ( i = 1:n ) a(i,i) = log( abs( a(i,i) ) )
```

Because a *forall-assignment* may be an array assignment, the pure function can have an array result. Such functions may be particularly helpful for performing row-wise or column-wise operations on an array. The next example illustrates this.

```
INTERFACE
  PURE FUNCTION f(x)
    REAL, DIMENSION(3) :: f,
    REAL, DIMENSION(3), INTENT(IN) :: x
  END FUNCTION f
END INTERFACE
REAL  v (3,10,10)
...
FORALL (i=1:10, j=1:10)  v(:,i,j) = f(v(:,i,j))
```

A limited form of MIMD parallelism can be obtained by means of branches within the pure procedure that depend on arguments associated with array elements or their subscripts when the function is called from a **FORALL**. This may sometimes provide an alternative to using sequences of masked **FORALL** or **WHERE** statements with their potential synchronization overhead. The next example suggests how this may be done.

```
REAL PURE FUNCTION f (x, i)
  REAL, INTENT(IN) :: x       ! associated with array element
  INTEGER, INTENT(IN) :: i    ! associated with array subscript
  IF (x > 0.0) THEN              ! content-based conditional
    f = x*x
  ELSE IF (i==1 .OR. i==n) THEN  ! subscript-based conditional
    f = 0.0
  ELSE
    f = x
  ENDIF
END FUNCTION

...

REAL a(n)
INTEGER i
  ...
FORALL (i=1:n)  a(i) = f( a(i), i)
```

Because pure procedures have no constraints on their internal control flow (except that they may not use the **STOP** statement), they also provide a means for encapsulating more complex operations than could otherwise be nested within a **FORALL**. For example, the fragment below performs an iterative algorithm on every element of an array. Note that different amounts of computation may be required for different inputs. Some machines may not be able to take advantage of this flexibility.

```
PURE INTEGER FUNCTION iter(x)
  COMPLEX, INTENT(IN) :: x
  COMPLEX xtmp
  INTEGER i
  i = 0
  xtmp = -x
  DO WHILE (ABS(xtmp).LT.2.0 .AND. i.LT.1000)
    xtmp = xtmp * xtmp - x
    i = i + 1
  END DO
  iter = i
END FUNCTION

  ...
FORALL (i=1:n, j=1:m)  ix(i,j) = iter(CMPLX(a+i*da,b+j*db))
```

4.3.4 Comments on Pure Procedures

Rationale.

The constraints for a pure procedure guarantee freedom from side-effects, thus ensuring that it can be invoked concurrently at each "element" of an array (where an "element" may itself be a data structure, including an array).

The constraints on pure procedures may appear complicated, but it is not necessary for a programmer to be intimately familiar with them. From the programmer's point of view, these constraints can be summarized as follows: a pure procedure must not contain any operation that could conceivably result in an assignment or pointer assignment to a global variable or `INTENT (IN)` dummy argument, or perform any I/O or `STOP` operation. Note the use of the word *conceivably*; it is not sufficient for a pure procedure merely to be side-effect free *in practice*. For example, a function that contains an assignment to a global variable but in a branch that is not executed in any invocation of the function is nevertheless not a pure function. The exclusion of functions of this nature is unavoidable if strict compile-time checking is to be used. In the choice between compile-time checking and flexibility, the HPF committee decided in favor of enhanced checking.

It is expected that most library procedures will conform to the constraints required of pure procedures (by the very nature of library procedures), and so can be declared pure and referenced in `FORALL` statements and constructs and within user-defined pure procedures. It is also anticipated that most library procedures will not reference global data, whose use may sometimes inhibit concurrent execution.

The constraints on pure procedures are limited to those necessary to check statically for freedom from side effects, processor independence, and for lack of saved internal state. Subject to these restrictions, maximum functionality has been preserved in the definition of pure procedures. This has been done to make function calls in `FORALL` as widely available as possible, and so that quite general library procedures can be classified as pure.

A drawback of this flexibility is that pure procedures permit certain features whose use may hinder, and in the worst case prevent, concurrent execution in `FORALL` (that is, such references may have to be implemented by sequentialization). Foremost among these features are the access of global data, particularly distributed global data, and the fact that the arguments and, for a pure function, the result may be pointers or data structures with pointer components, including recursive data structures such as lists and trees. The programmer should be aware of the potential performance penalties of using such features.

(*End of rationale.*)

4.4 The INDEPENDENT Directive

The `INDEPENDENT` directive can precede a `DO` loop or `FORALL` statement or construct. It asserts to the compiler that the operations in the following `FORALL` statement or construct or iterations in the following `DO` loop may be executed independently—that is, in any order, or interleaved, or concurrently—without changing the semantics of the program.

The INDEPENDENT directive precedes the DO loop or FORALL for which it is asserting behavior, and is said to apply to that loop or FORALL. The syntax of the INDEPENDENT directive is

H413 *independent-directive* **is** INDEPENDENT [, *new-clause*]

H414 *new-clause* **is** NEW (*variable-list*)

Constraint: The first non-comment line following an *independent-directive* must be a *do-stmt*, *forall-stmt*, or a *forall-construct*.

Constraint: If the NEW option is present, then the directive must apply to a DO loop.

Constraint: A *variable* named in the NEW option or any component or element thereof must not:

- Be a pointer or dummy argument; nor
- Have the SAVE or TARGET attribute.

When applied to a DO loop, an INDEPENDENT directive is an assertion by the programmer that no iteration can affect any other iteration, either directly or indirectly. The following operations define such interference:

- Any two operations that assign to the same atomic object (defined in Section 4.1.2) interfere with each other. (Note the NEW clause below, however.)
- An operation that assigns to an atomic object interferes with any operation that uses the value of that object. (Note the NEW clause below, however.)

  Rationale. These are the classic Bernstein [5] conditions to enable parallel execution. Note that two assignments *of the same value* to a variable interfere with each other and thus an INDEPENDENT loop with such assignments is not HPF-conforming. This is not allowed because such overlapping assignments are difficult to support on some hardware, and because the given definition was felt to be conceptually clearer. Similarly, it is not HPF-conforming to assert that assignment of multiple values to the same location is INDEPENDENT, even if the program logically can accept any of the possible values. In this case, both the "conceptually clearer" argument and the desire to avoid nondeterministic behavior favored the given solution. (*End of rationale.*)

- Any transfer of control to a branch target statement outside the body of the loop interferes with all other operations in the loop.
- Any execution of an EXIT, STOP, or PAUSE statement interferes with all other operations in the loop.

  Rationale. Branching (by GOTO or ERR= branches in I/O statements) implies that some iterations of the loop are not executed, which is drastic interference with those computations. The same is true for EXIT and the other statements. Note that these conditions do not restrict procedure calls in INDEPENDENT loops, except to disallow taking alternate returns to statements outside the loop. (*End of rationale.*)

- A **READ** operation assigns to the objects in its *input-item-list*; a **WRITE** or **PRINT** operation uses the values of the objects on its *output-item-list*. I/O operations may interfere with other operations (including other I/O operations) as per the conditions above.
- An internal **READ** operation uses its internal file; an internal **WRITE** operation assigns to its internal file. These uses and assignments may interfere with other operations as outlined above.
- Any two file I/O operations except **INQUIRE** associated with the same file or unit interfere with each other. Two **INQUIRE** operations do not interfere with each other; however, an **INQUIRE** operation interferes with any other I/O operation associated with the same file.

  Rationale. Because Fortran carefully defines the file position after a data transfer or file positioning statement, these operations affect the global state of a program. (Note that file position is defined even for direct access files.) Multiple non-advancing data transfer statements affect the file position in ways similar to multiple assignments of the same value to a variable, and is disallowed for the same reason. Multiple **OPEN** and **CLOSE** operations affect the status of files and units, which is another global side effect. **INQUIRE** does not affect the file status, and therefore does not affect other inquiries. However, other file operations may affect the properties reported by **INQUIRE**. (*End of rationale.*)

- Any data realignment or redistribution performed in the loop interferes with any access to or any other realignment of the same data.

  Rationale. **REALIGN** and **REDISTRIBUTE** may change the processor storing a particular array element, which interferes with any assignment or use of that element. Similarly, multiple remapping operations may cause the same element to be stored in multiple locations. (*End of rationale.*)

Note that all of these describe interfering behavior; they do not disallow specific syntax. Statements that appear to violate one or more of these restrictions are allowed in an **INDEPENDENT** loop, if they are not executed due to control flow. These restrictions allow an **INDEPENDENT** loop to be executed safely in parallel if computational resources are available. The directive is purely advisory and a compiler is free to ignore it if it cannot make use of the information.

The **NEW** option modifies the **INDEPENDENT** directive and all surrounding **INDEPENDENT** directives by asserting that those assertions would be true *if* new objects were created for the named variables for each iteration of the **DO** loop. Thus, variables named in the *new-clause* behave as if they were private to the body of the **DO** loop. More formally, it asserts that the remainder of program execution is unaffected if all variables in the *variable-list* and any variables associated with them were to become undefined immediately before execution of every iteration of the loop, and also become undefined immediately after the completion of each iteration of the loop.

Advice to implementors.

The wording here is similar to the treatment of realignment through pointers in Section 3.6. As with that section, it may be reworded if HPF directives are absorbed as actual Fortran statements.

(*End of advice to implementors.*)

Rationale. NEW variables provide the means to declare temporaries in INDEPENDENT loops. Without this feature, many conceptually independent loops would need substantial rewriting (including expansion of scalars into arrays) to meet the rather strict requirements described above. Note that a temporary need only be declared NEW at the innermost lexical level at which it is assigned, since all enclosing INDEPENDENT assertions must take that NEW into account. Note also that index variables for nested DO loops must be declared NEW; the alternative was to limit the scope of an index variable to the loop itself, which changes Fortran semantics. FORALL indices, however, are restricted by the semantics of the FORALL; they require no NEW declarations. (*End of rationale.*)

Advice to users. Section 4.4.1 contains several examples of the syntax and semantics of INDEPENDENT applied to DO loops. (*End of advice to users.*)

The interpretation of INDEPENDENT for FORALL is similar to that for DO: it asserts that no combination of the indexes that INDEPENDENT applies assigns to an atomic storage unit that is read by another combination. (Note that an HPF FORALL statement or construct does not allow exits from the construct, etc.) A DO and a FORALL with the same body are equivalent if they both have the INDEPENDENT directive. This is illustrated in Section 4.4.2.

4.4.1 Examples of INDEPENDENT

```
!HPF$ INDEPENDENT
DO i = 2, 99
  a(i) = b(i-1) + b(i) + b(i+1)
END DO
```

This is one of the simplest examples of an INDEPENDENT loop. (For simplicity, all examples in this section assume there is no storage or sequence association between any variables used in the code.) Every iteration assigns to a different location in the a array, thus satisfying the first condition above. Since no elements of a are used on the right-hand side, no location that is assigned in the loop is also read, thus satisfying the second condition. Note, however, that many elements of b are used repeatedly; this is allowed by the definition of INDEPENDENT. The other conditions relate to constructs not used in the loop. In this example, the assertion is true regardless of the values of the variables involved.

```
!HPF$ INDEPENDENT
FORALL ( i=2:n ) a(i) = b(i-1) + b(i) + b(i+1)
```

This example is equivalent in all respects to the first example.

```
!HPF$ INDEPENDENT
DO i=1, 100
  a(p(i)) = b(i)
END DO
```

This INDEPENDENT directive asserts that the array p does not have any repeated entries (else they would cause interference when a was assigned). The DO loop is therefore equivalent to the Fortran 90 statement

```
a(p(1:100)) = b(1:100)

!HPF$ INDEPENDENT, NEW (i2)
DO i1 = 1,n1
 !HPF$ INDEPENDENT, NEW (i3)
 DO i2 = 1,n2
  !HPF$ INDEPENDENT, NEW (i4)
  DO i3 = 1,n3
   DO i4 = 1,n4   ! The inner loop is NOT independent!
    a(i1,i2,i3) = a(i1,i2,i3) + b(i1,i2,i4)*c(i2,i3,i4)
   END DO
  END DO
 END DO
END DO
```

The inner loop is not independent because each element of a is assigned repeatedly. However, the three outer loops are independent because they access different elements of a. The NEW clauses are required, since the inner loop indices are assigned and used in different iterations of the outermost loops.

```
!HPF$ INDEPENDENT, NEW (j)
DO i = 2, 100, 2
  !HPF$ INDEPENDENT, NEW(vl, vr, ul, ur)
  DO j = 2 , 100, 2
    vl = p(i,j) - p(i-1,j)
    vr = p(i+1,j) - p(i,j)
    ul = p(i,j) - p(i,j-1)
    ur = p(i,j+1) - p(i,j)
    p(i,j) = f(i,j) + p(i,j) + 0.25 * (vr - vl + ur - ul)
  END DO
END DO
```

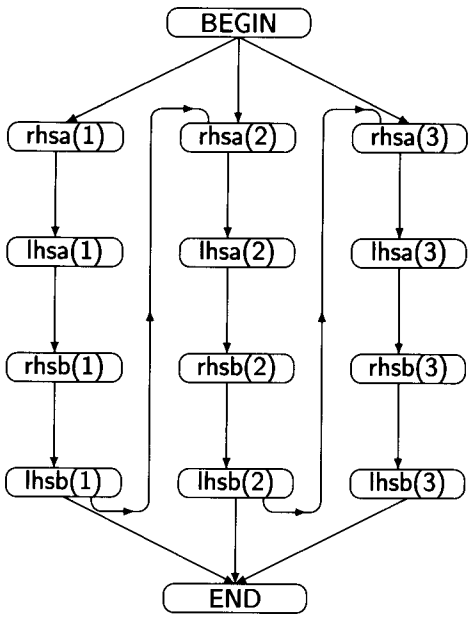
Without the NEW option on the j loop, neither loop would be independent, because an interleaved execution of loop iterations might cause other values of vl, vr, ul, and ur to be used in the assignment of $p(i,j)$ than those computed in the same iteration of the loop. The NEW option, however, specifies that this is not true if distinct storage units are used in each iteration of the loop. Using this implementation makes iterations of the loops independent of each other. Note that there is no interference due to accesses of the array p because of the stride of the DO loop (i.e. i and j are always even, therefore $i-1$, etc. are always odd.)

```
!HPF$ INDEPENDENT
      DO i = 1, 10
        WRITE (iounit(i),100) a(i)
      END DO
100   FORMAT ( F10.4 )
```

If $iounit(i)$ evaluates to a different value for every $i \in \{1,\ldots,10\}$, then the loop writes to a different I/O unit (and thus a different file) on every iteration. The loop is then properly described as independent. On the other hand, if $iounit(i) = 5$ for all i, then the assertion is in error and the loop is not HPF-conforming.

4.4.2 Visualization of INDEPENDENT Directives

```
DO i = 1, 3
  lhsa(i) = rhsa(i)
  lhsb(i) = rhsb(i)
END DO
```




```
FORALL ( i = 1:3 )
  lhsa(i) = rhsa(i)
  lhsb(i) = rhsb(i)
END FORALL
```

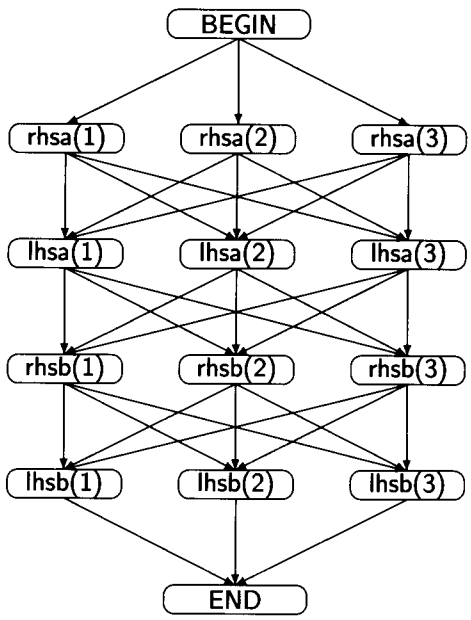
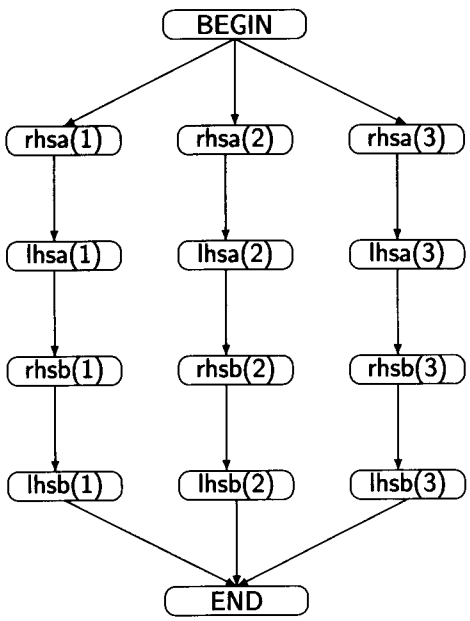


Figure 4.1: Dependences in DO and FORALL without INDEPENDENT assertions

Graphically, the INDEPENDENT directive can be visualized as eliminating edges from a precedence graph representing the program. Figure 4.1 shows some of the dependences that may normally be present in a DO and a FORALL. (Most of the transitive dependences are not shown.) An arrow from a left-hand side node (for example, "lhsa(1)") to a right-hand side node ("rhsb(1)") means that the right-hand side computation might use values assigned in the left-hand side node; thus the right-hand side must be computed after the left-hand side completes its store. Similarly, an arrow from a right-hand side node to a left-hand side node means that the left-hand side may overwrite a value needed by the right-hand side computation, again forcing an ordering. Edges from the "BEGIN" and to the "END" nodes represent control dependences. The INDEPENDENT directive asserts that the only dependences that a compiler need enforce are those in Figure 4.2. That is, the programmer who uses INDEPENDENT is certifying that if the compiler enforces only these edges, then the resulting program will be equivalent to the one in which all the edges are present. Note that the set of asserted dependences is identical for INDEPENDENT DO and FORALL constructs.

The compiler is justified in producing a warning if it can prove that one of these assertions is incorrect. It is not required to do so, however. A program containing any false assertion of this type is not HPF-conforming, thus is not defined by HPF, and the compiler may take any action it deems appropriate.

```
!HPF$ INDEPENDENT
DO i = 1, 3
  lhsa(i) = rhsa(i)
  lhsb(i) = rhsb(i)
END DO
```




```
!HPF$ INDEPENDENT
FORALL ( i = 1:3 )
  lhsa(i) = rhsa(i)
  lhsb(i) = rhsb(i)
END FORALL
```

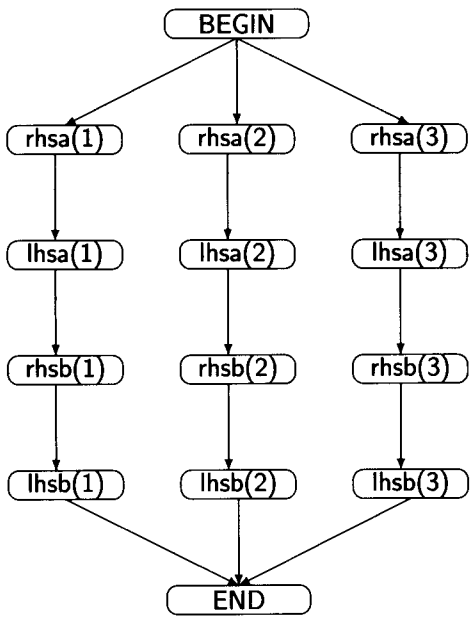


Figure 4.2: Dependences in DO and FORALL with INDEPENDENT assertions