CLAW language specification

Center for Climate System Modeling (C2SM)
http://www.c2sm.ethz.ch
Valentin Clement

Last updated: October 24, 2016

v0.3a

Contents

1 General information about the CLAW language

The directives are either local or global.

- Local directive: those directives have a limited impact on a local block of code (for example, only in a subroutine or only in a do statement)

- Global directive: those directives can have an impact on the whole program.

The CLAW transformations are separated in the followings sections:

- CLAW abstraction transformations
- Loop transformations
- Accelerator abstraction/helper transformations
- Utility transformations

1.1 Line continuation

CLAW directives can be defined on several lines with the continuation line character. The syntax is described as follows:

```
!$claw directive clause &
!$claw clause
```

1.2 Interpretation order of the CLAW directives

The CLAW transformations can be combined together. For example, `loop-fusion` and `loop-interchange` can be used together in a group of nested loops.

The default interpretation order of the directives is the following:

1. `ignore`
2. `remove`
3. `parallelize`
4. `parallelize forward`
5. `array-transform`
6. `loop-extract`
7. `loop-fusion`
8. `loop-hoist`
9. `loop-interchange`

10. `on-the-fly`
11. `kcache`
12. formatting transformation (internal use transformation)

Users must be aware that transformations are applied sequentially and therefore, a transformation can be performed on code that has already been transformed.

1.3 Accelerator compile guard

As CLAW generates accelerator directives, it might be useful sometimes to protect the compilation of source code including accelerator directives as it might be wrong if the CLAW Fortran compiler didn't process it.

For this reason, the CLAW language has a compile guard that is removed by the CLAW Fortran compiler once it applies the transformations. This compile guard is designed to make the compilation failed when transformations are not applied and accelerator directives are present in the source code.

This compile guard is shown in the listing below. It uses the target accelerator language with a specific claw directive. Example for OpenACC on line 2 and OpenMP on line 3.

```
1 !$<accelerator language prefix> claw-guard
2 !$acc claw-guard
3 !$omp claw-guard
```

2 CLAW abstraction

2.1 Column model abstraction

The `parallelize` directive is used to parallelize a column based algorithm. In weather prediction models, the physical parametrizations are column independent problems. This means that the algorithm to compute the different output fields, can be defined with as a single loop iterating over the levels of the column. The `parallelize` directive is then used to parallelize efficiently this column computation over the domain (horizontal dimensions). The directive is local to a subroutine. Therefore, as a pre-condition, the algorithm for the column computation must be enclosed in a subroutine.

```
!$claw define dimension dim_id(lower_bound:upper_bound) &
[!$claw define dimension dim_id(lower_bound:upper_bound) &] ...
!$claw parallelize [data(data(var_1[,var_2] ...)] [over (dim_id|:[,dim_id|:]
    ...)]
```

Options and details

- *define dimension*: Define a new dimension on which the column model will be parallelized. The lower bound and upper bound of the defined dimension can be either an integer constant or an integer variable. If a variable is given, it will be added in-order to the signature of the subroutine as an `INTENT(IN)` parameter.
- *data (optional)*: Define a list of variables to be promoted and bypass the automatic promotion.
- *over (optional)*: Define the location of the new dimensions for the promotion of the variables. The : sign reflects the position of the current dimensions before the promotion.

If the `data` clause is omitted, the automatic promotion of variables is done as follows: Array variables declared with the `INOUT` or `OUT` intents are automatically promoted with the defined dimensions. The dimensions are added on the left-side following the the definition order if there is more than one. In function of the target, intermediate scalar or array variables may also be promoted. Scalar/array variables placed on the left hand-side of an assignment statement will be promoted if the right hand-side references any variables already promoted. If the `over` clause is omitted, the references are promoted with the defined dimensions added at the beginning from left to right. The `over` clause allows to change this behavior and to place the new dimensions where needed.

As the subroutine/function signature is updated by the `parallelize` directive with new dimensions, the call graph that leads to this subroutine/function must be updated as well. For this purpose, the clause `forward` allows to replicated the changes along the call graph. Each subroutine/function along the call graph must be decorated with this directive.

```
! Call of a parallelized subroutine
!$claw parallelize forward
CALL one_column_fct(x,y,z)

! Call of a parallelized function
!$claw parallelize forward
result = one_column_fct(x,y,z)
```

The different variables declared in the current function/subroutine might be promoted if the call requires it. In this case, the array references in the function/subroutine are updated accordingly. This can lead to extra promotion of local scalar variables or arrays when needed.

The first function call in the call graph might be an iteration over several column to reproduce the algorithm on the grid for testing purpose. If the directive with the `forward` clause is placed just before one or several nested do statements with a function call, the corresponding do statements will be removed and the function call updated accordingly.

```
!$claw parallelize forward
DO i=istart, iend
  CALL one_columne_fct(x,y(i,:),z(i,:))
END DO
```

Code example
Simple example of a column model wrapped into a subroutine and parallelize with CLAW.

Original code

```
PROGRAM model
  USE mo_column, ONLY: compute_column

  REAL, DIMENSION(20,60) :: q, t  ! Fields as declared in the whole model
  INTEGER :: nproma, nz           ! Size of array fields
  INTEGER :: p                    ! Loop index

  nproma = 20
  nz = 60

  DO p = 1, nproma
    q(p,1) = 0.0
    t(p,1) = 0.0
  END DO

  !$claw parallelize forward
  DO p = 1, nproma
    CALL compute_column(nz, q(p,:), t(p,:))
  END DO

  PRINT*,SUM(q)
  PRINT*,SUM(t)
END PROGRAM model
```

```
MODULE mo_column
  IMPLICIT NONE
CONTAINS
  ! Compute only one column
  SUBROUTINE compute_column(nz, q, t)
    IMPLICIT NONE

    INTEGER, INTENT(IN)   :: nz   ! Size of the array field
    REAL, INTENT(INOUT)   :: t(:) ! Field declared as one column only
    REAL, INTENT(INOUT)   :: q(:) ! Field declared as one column only
    INTEGER :: k                  ! Loop index
    REAL :: c                     ! Coefficient
    REAL :: d                     ! Intermediate variable

    ! CLAW definition

    ! Define one dimension that will be added to the variables defined in the
    ! data clause.
    ! Apply the parallelization transformation on this subroutine.

    !$claw define dimension proma(1:nproma) &
```

```
    !$claw parallelize

    c = 5.345
    DO k = 2, nz
      t(k) = c * k
      d    = t(k)**2
      q(k) = q(k - 1)  + t(k) * c
    END DO
    q(nz) = q(nz) * c
  END SUBROUTINE compute_column
END MODULE mo_column
```

Transformed code
Only the code of the module is shown as it is the only one which changes.

CLAW generated code for GPU target with OpenACC accelerator directive language.

```
MODULE mo_column

CONTAINS
 SUBROUTINE compute_column ( nz , q , t , nproma )
  INTEGER , INTENT(IN) :: nz
  REAL , INTENT(INOUT) :: t ( : , : )
  REAL , INTENT(INOUT) :: q ( : , : )
  INTEGER :: k
  REAL :: c
  REAL :: d
  INTEGER , INTENT(IN) :: nproma
  INTEGER :: proma

!$acc data present(t,q,nproma,nz)
!$acc parallel private(k,proma,d,c)
!$acc loop
  DO proma = 1 , nproma , 1
   c = 5.345
!$acc loop seq
   DO k = 2 , nz , 1
    t ( proma , k ) = c * k
    d = t ( proma , k ) ** ( 2 )
    q ( proma , k ) = q ( proma , k - 1 ) + t ( proma , k ) * c
   END DO
   q ( proma , nz ) = q ( proma , nz ) * c
  END DO
!$acc end parallel
!$acc end data
 END SUBROUTINE compute_column

END MODULE mo_column
```

CLAW generated code for CPU target with OpenMP accelerator directive language.

```
MODULE mo_column

CONTAINS
 SUBROUTINE compute_column ( nz , q , t , nproma )
  INTEGER , INTENT(IN) :: nz
  REAL , INTENT(INOUT) :: t ( : , : )
  REAL , INTENT(INOUT) :: q ( : , : )
  INTEGER :: k
  REAL :: c
  REAL :: d ( 1 : nproma )
  INTEGER , INTENT(IN) :: nproma
  INTEGER :: proma

!$omp parallel
  c = 5.345
  DO k = 2 , nz , 1
!$omp do
   DO proma = 1 , nproma , 1
    t ( proma , k ) = c * k
   END DO
!$omp end do
!$omp do
   DO proma = 1 , nproma , 1
    d ( proma ) = t ( proma , k ) ** ( 2 )
   END DO
!$omp end do
!$omp do
   DO proma = 1 , nproma , 1
    q ( proma , k ) = q ( proma , k - 1 ) + t ( proma , k ) * c
   END DO
!$omp end do
  END DO
!$omp do
  DO proma = 1 , nproma , 1
   q ( proma , nz ) = q ( proma , nz ) * c
  END DO
!$omp end do
!$omp end parallel
 END SUBROUTINE compute_column

END MODULE mo_column
```

Compilation with the CLAW Fortran Compiler

```
# For GPU target with OpenACC directive
clawfc --target=gpu --directive=openacc -o mo_column.gpu_acc.f90 mo_column.f90
clawfc -o main.claw.f90 main.f90

# For CPU target with OpenMP directive
```

```
clawfc --target=gpu --directive=openmp -o mo_column.cpu_omp.f90 mo_column.f90
clawfc -o main.claw.f90 main.f90
```

2.2 K caching (column caching)

```
!$claw kcache data(var_1[,var_2] ...) [offset(offset_1[,offset_2] ...)]] [init]
     [private]
```

In memory-bound problem, it might be useful to cache array values used several times during the current do statement iteration or for the next one.

The **kcache** directive is applied in a limited do statement block. It will cache the corresponding assigned value and update the array index in the block according to the given plus/minus offsets. If the array referenced in the data clause is assigned in the block, the caching will take its place (see example 2.2). Otherwise, an assignment statement is created to start the caching (see example A.1).

If the offset values are omitted, there are inferred from the dimension of the variable to be cached and set to 0.

Options and details

- *data*: The data clause specifies which array will be impacted by the column caching transformation. List of array identifiers.
- *offset*: Integer value separated by a comma that represents the offset at each dimension.
- *init*: If the *init* clause is specified, the cache variable will be initialized with the corresponding array value at the given offset during the 1st loop iteration.
- *private*: it declares that a copy of each item on the list will be created for each parallel gang on the accelerator. The list is the one specified on the *data* clause.

Code example

Where caching takes assignment place and `init` clause.

Original code

```
DO j1 = k1start, k1end
  array_u6(j1,k3end) = pc1(j1,kend) * p2(j1)
ENDDO

DO j3 = k3start + 1, k3end
  DO j1 = k1start, k1end
    ! More computation here with others variables

```

```
    value1 = 1.0_dp / (1.0_dp - p2(j1) * (pc2(j1,j3) * array6(j1,j3-1)  &
                                   +  pc2(j1,j3) * array8(j1,j3-1) ))
    value2 = p2(j1) * (pc2(j1,j3) * array6(j1,j3-1)                     &
                     + pc2(j1,j3) * array8(j1,j3-1))
    value6 = p1(j1) * (pc2(j1,j3) * array6(j1,j3-1)                     &
                      +pc2(j1,j3) * array8(j1,j3-1))
        !$claw kcache data(array6) offset(0 -1) init
    array6(j1,j3) = p2(j1) * pc1(j1,j3)                                 &
                    + value2 * array8(j1,j3) + value6 * array8(j1,j3)
    array8(j1,j3) = p2(j1) * pc1(j1,j3)                                 &
                    + value6 * array2(j1,j3) + value7 * array4(j1,j3)
  END DO
END DO
```

Transformed code

```
DO j1 = k1start, k1end
  array_u6(j1,k3end) = pc1(j1,kend) * p2(j1)
ENDDO

DO j1 = k1start, k1end
  DO j3 = k3start + 1, k3end
    IF (j3 = k3start + 1) THEN
      array6_k_m1 = array6(j1,j3-1)
      array8_k_m1 = array8(j1,j3-1)
    END IF
    ! More computation here with others variables

    value1 = 1.0_dp / (1.0_dp - p2(j1) * (pc2(j1,j3) * array6_k_m1    &
                                   +  pc2(j1,j3) * array8_k_m1 ))
    value2 = p2(j1) * (pc2(j1,j3) * array6_k_m1                       &
                     + pc2(j1,j3) * array8_k_m1
    value6 = p1(j1) * (pc2(j1,j3) * array6_k_m1                       &
                      +pc2(j1,j3) * array8_k_m1
        !$claw kcache data(array6, array8) offset(0 -1) init
        array6_k_m1 = p2(j1) * pc1(j1,j3)                               &
                    + value2 * array8(j1,j3) + value6 * array8(j1,j3)
    array6(j1,j3) = array6_k_m1
    array8_k_m1 = p2(j1) * pc1(j1,j3)                                 &
                    + value6 * array2(j1,j3) + value7 * array4(j1,j3)
    array8(j1,j3) = array8_k_m1
  END DO
END DO
```

More code examples in the appendix. Example with assignment created during the transformation (see example A.1). Example with `init` and `private` clause (see example A.2).

2.3 On-the-fly computation (array access to function call)

```
!$claw call array_name=function_call(arg_list)
```

Sometimes, replacing access to pre-computed arrays with computation on-the-fly can increase the performance. It can reduce the memory access for memory-bound kernel and exploit some unused resources to execute the computation.

This transformation is local to the current do statement. Array access from the directive will be replaced.

Options and details

- *array_name*: Array identifier. References to this array will be replaced by the function call in the current structured block.
- *function_call*: Name of the function provided by the user to compute the value. Return value must be of the same type as accessed value.
- *arg_list*: Comma separated list of arguments to be passed to the function call.

3 Loop transformation

3.1 Loop interchange/reordering

```
!$claw loop-interchange [(induction_var[, induction_var] ...)]
```

Loop reordering is a common transformation applied on do statements when adding parallelization. This transformation is mainly used to improve the data locality and avoid the need to transpose the arrays.

The **loop-interchange** directive must be placed just before the first do statement composing the nested do statements group. With a group of two nested do statements, the induction variable and iteration range of the first statement is swap with the information of the second statement (see example 3.1).

If the list of induction variables is specified, the do statements are reordered according to the order defined in the list (see example A.3).

Options and details

1. *induction_var*: induction variable of the do statement. The list gives the new order of the do statement after the transformation. For group of 2 nested do statements, this information is optional.

Behavior with other directives
When the do statements to be reordered are decorated with additional directives, those directives stay in place during the code transformation. In other words, they are not reordered together with the do statements iteration ranges (see example A.4).

Limitations
Currently, the **loop-interchange** directive is limited to 3 level of nested do statements. More nested levels can be declared but the transformation is limited to the first 3 levels from the directive declaration.

Code example
Example with 2 levels of loop.

Original code

```
!$claw loop-interchange
DO i=1, iend
  DO k=1, kend
    ! loop body here
  END DO
END DO
```

Transformed code

```
! CLAW transformation (loop-interchange i < -- > k)
DO k=1, kend
  DO i=1, iend
    ! loop body here
  END DO
END DO
```

More code examples in the appendix. Example with 3 nested statements (see example A.3). Example with others directives (see example A.4).

3.2 Loop jamming/fusion

```
!$claw loop-fusion [group(group_id)] [collapse(n)]
```

Loop jamming or fusion is used to merge 2 or more do statements together. Sometimes, the work performed in a single do statement is too small to create significant impact on performance when it is parallelized. Merging some do statements together create bigger blocks (kernels) to be parallelized.

If the *group* clause is not specified, all the do statements decorated with the directive in the same structured block will be merged together as a single do statement (see code example in this section).

If the *group* clause is specified, the loops are merged in-order with do statements sharing the same group in the current structured block (see example A.5).

The *collapse* clause is used to specify how many tightly nested do statements are associated with the **loop-fusion** construct (see example A.7). The argument to the *collapse* clause is a constant positive integer. If the *collapse* clause is not specified, only the do statement that follows immediately the directive is associated with the **loop-fusion** construct.

Options and details

- *group_id*: An identifier that represents a group of do statement in a structured block (see example A.5).
- *n*: A constant positive integer.

Behavior with other directives

When the do statement to be merged are decorated with other directives, only the directives on the first do statement of the merge group are kept in the transformed code.

Limitations

All the do statement within a group must share the same iteration range. If the *collapse* clause is specified, the do statements must share the same iteration range at the corresponding depth (see example A.7).

Code example

Example without clauses.

Original code

```
DO k=1, iend
  !$claw loop-fusion
  DO i=1, iend
    ! loop #1 body here
  END DO

  !$claw loop-fusion
  DO i=1, iend
    ! loop #2 body here
  END DO
END DO
```

Transformed code

```
DO k=1, iend
  ! CLAW transformation (loop-fusion same block group)
  DO i=1, iend
    ! loop #1 body here
    ! loop #2 body here
  END DO
END DO
```

More code examples in the appendix. Example with the `group` clause (see example A.5). Example with others directives (see example A.6). Example with the `collapse` clause (see example A.7).

3.3 Loop extraction

```
!$claw loop-extract range(range) [map(var[,var]...:mapping[,mapping]...) [map(var[,var]...:mapping[,mapping]...)] ...] [fusion [group(group_id)]] [parallel] [acc(directives)]
```

Loop extraction can be performed on a subroutine/function call. The do statement corresponding to the defined iteration range is extracted from the subroutine/function and is wrapped around the subroutine/function call. In the transformation, a copy of the subroutine/function is created with the corresponding transformation (demotion) for the parameters.

Options and details

1. *range*: Correspond to the iteration range of the loop to be extracted. Notation `i = istart , iend, istep`. Step value is optional.
2. *map*: Define the mapping of variable that are demoted during the loop extraction. As seen in the example 3.3, the two parameters (1 dimensional array) are mapped to a scalar with the induction variable *i*.
   (a) The *var* can be defined as two parts variable (e.g. `a/a1`). The first part is the function call part and refers to the variable as it is defined in the function call. The second part is the function definition part and refers to the name of the variable to be mapped as it defined in the function declaration. If a *var* is defined as a single part variable, the same name is used for both the function call and function definition part.
   (b) The *mapping* can be defined as two parts variable (e.g. `i/i1`). The first part is the function call part and refers to the mapping variable as it is defined in the function call. The second part is the function definition part and refers to the name of the mapping variable as it defined in the function declaration. If a *mapping* is defined as a single part mapping variable, the same name is used for both the function call and function definition part.
3. *fusion*: Allow the extracted loop to be merged with other loops. Options are identical with the `loop-fusion` directive.
4. *parallel*: Wrap the extracted loop in a parallel region.
5. *acc*: Add the accelerator directives to the extracted loop.

If the directive **loop-extract** is used for more than one call to the same subroutine, the extraction can generate 1 to N dedicated subroutines.

Behavior with other directives
Accelerator directives are generated by the transformation. If the function call was decorated with accelerator directives prior to the transformation, those directives stay in place.

Code example
Example without additional clauses.

Original code

```
PROGRAM main
  !$claw loop-extract(i=istart,iend) map(value1,value2:i)
  CALL xyz(value1, value2)
END PROGRAM main

SUBROUTINE xyz(value1, value2)
  REAL, INTENT (IN) :: value2(x:y), value2(x:y)

  DO i = istart, iend
    ! some computation with value1(i) and value2(i) here
  END DO
END SUBROUTINE xyz
```

Transformed code

```
PROGRAM main
  !CLAW extracted loop
  DO i = istart, iend
    CALL xyz_claw(value1(i), value2(i))
  END DO
END PROGRAM main

SUBROUTINE xyz(value1, value2)
  REAL, INTENT (IN) :: value2(x:y), value2(x:y)

  DO i = istart, iend
    ! some computation with value1(i) and value2(i) here
  END DO
END SUBROUTINE xyz

!CLAW extracted loop new subroutine
SUBROUTINE xyz_claw(value1, value2)
  REAL, INTENT (IN) :: value1, value2
  ! some computation with value1 and value2 here
END SUBROUTINE xyz_claw
```

More code examples in the appendix. Example with the `fusion` clause (see example A.8). Example with a more complicated mapping (see example A.9).

3.4 Loop hoisting

```
!$claw loop-hoist(induction_var[[, induction_var] ...]) [reshape(array_name(
    target_dimension[,kept_dimensions]))] [interchange [(induction_var[[,
    induction_var] ...])]]

  ! structured block of code

!$claw end loop-hoist
```

The **loop-hoist** directive allows nested loops in a defined structured block to be merged together and to hoist the beginning of those nested loop just after the directive declaration. Loops with different lower-bound indexes can also be merged with the addition of an `IF` statement. This feature works only when the lower-bound are integer constant.

Options and details

1. *induction_var*: List of induction variables of the do statements to be hoisted.
2. *interchange*: Allow the group of hoisted loops to be reordered. Options are identical with the **loop-interchange** directive.
3. *reshape*: Reshape arrays to scalar or to array with fewer dimensions. The original declaration is replaced with the corresponding demoted declaration in the current block (function/module).
   (a) *array_name*: Name of the array type to be reshapped.
   (b) *target_dimension*: Number of dimension after reshapaing. 0 indicates that the type is reshapped to a scalar.
   (c) *kept_dimension*: comma separated list of integer that indicates which dimensions are kept when the array is not totally demoted to scalar. Dimension index starts at 1.

Code example

Example without clauses.

Original code

```
!$acc parallel loop gang vector collapse(2)
DO jt=1,jtend
  !$claw loop-hoist(j,i) interchange
  IF ( .TRUE. ) CYCLE
    ! outside loop statement
  END IF
  DO j=1,jend
    DO i=1,iend
      ! first nested loop body
    END DO
```

```
  END DO
  DO j=2,jend
    DO i=1,iend
      ! second nested loop body
    END DO
  END DO
  !$claw end loop-hoist
END DO
!$acc end parallel
```

Transformed code

```
!$acc parallel loop gang vector collapse(2)
DO jt=1,jtend
  DO i=1,iend
    DO j=1,jend
      IF ( .TRUE. ) CYCLE
        ! outside loop statement
      END IF
      ! first nested loop body
      IF(j>1) THEN
        ! second nested loop body
      END IF
    END DO
  END DO
END DO
!$acc end parallel
```

4 Accelerator abstractions/helpers

4.1 Array notation to do statements

```
!$claw array-transform [induction(name [[,] name]...)] [fusion [group(group_id)
    ]] [parallel] [acc([clause [[,] clause]...])]

  ! array notation assignment(s)

[!$claw end remove]
```

Computations using the array notation are not suitable to be parallelized with language like OpenACC. The **array-transform** directive allows to transform those notation with the corresponding do statements which are more suitable for parallelization.

The goal of this directive is to pass from an array notation assignment like this:

```
!$claw array-transform
A(1:n) = A(1+m:n+m) + B(1:n) * C(n+1:n+n)
```

To a do statement statement like this:

```
DO i=1,n
  A(i) = A(i+m) + B(i) * C(n+i)
END DO
```

If the directive is used as a block directive, the assignments are wrapped in a single do statement if their induction range match.

```
!$claw array-transform
A(1:n) = A(1+m:n+m) + B(1:n) * C(n+1:n+n)
B(1:n) = B(1:n) * 0.5
!$claw end array-transform
```

To

```
DO i=1,n
  A(i) = A(i+m) + B(i) * C(n+i)
  B(i) = B(i) * 0.5
END DO
```

Options and details

1. *induction*: Allow to name the induction variable created for the do statement.
2. *fusion*: Allow the extracted loop to be merged with other loops. Options are identical with the **loop-fusion** directive
3. *parallel*: Wrap the extracted loop in a parallel region.
4. *acc*: Define accelerator clauses that will be applied to the generated loops.

Behavior with other directives
Directives declared before the **array-transform** directive will be kept in the generated code.

Code example
Example with array notation following the pragma statement.

Original code

```
SUBROUTINE vector_add
  INTEGER :: i = 10
  INTEGER, DIMENSION(0:9) :: vec1

```

```
  !$claw array-transform
  vec1(0:i) = vec1(0:i) + 10;
END SUBROUTINE vector_add
```

Transformed code

```
SUBROUTINE vector_add
  INTEGER :: claw_i
  INTEGER :: i = 10
  INTEGER, DIMENSION(0:9) :: vec1

  ! CLAW transformation array notation to do loop
  DO claw_i = 0, i
    vec1(claw_i) = vec1(claw_i) + 10;
  END DO
END SUBROUTINE vector_add
```

More code examples in the appendix. Example with the induction and acc (see example A.10). Example with fusion clause (see example A.11). Example with 2-dimensional arrays (see example A.12).

5 Utilities

5.1 Remove

```
!$claw remove

  ! code block

[!$claw end remove]
```

The **remove** directive allows the user to remove section of code during the transformation process. This code section is lost in the transformed code.

Options and details
If the directive is directly followed by a structured block (`IF` or `DO`), the end directive is not mandatory (see example 5.1). In any other cases, the end directive is mandatory.

Code example

Original code

```
DO k=1, kend
  DO i=1, iend
    ! loop #1 body here
```

```
  END DO

  !$claw remove
  IF (k > 1) THEN
    PRINT*, k
  END IF

  DO i=1, iend
    ! loop #2 body here
  END DO
END DO
```

Transformed code

```
DO k=1, kend
  DO i=1, iend
    ! loop #1 body here
  END DO

  DO i=1, iend
    ! loop #2 body here
  END DO
END DO
```

More code examples in the appendix. Example with block remove (see example A.13).

5.2 Ignore

```
!$claw ignore

  ! code block

!$claw end ignore
```

The `ignore` directive allows the user to ignore section of code for the transformation process. This code section is retrieved in the transformed code. Any CLAW directives in an ignored code section is also ignored.

Code example
In the example below, the first remove block is ignored and the second is not.

Original code

```
PROGRAM testignore

  !$claw ignore
  !$claw remove
```

```
  PRINT*,'These lines'
  PRINT*,'are ignored'
  PRINT*,'by the CLAW compiler'
  PRINT*,'but kept in the final transformed code'
  PRINT*,'with the remove directives.'
  !$claw end remove
  !$claw end ignore

  !$claw remove
  PRINT*,'These lines'
  PRINT*,'are not ignored.'
  !$claw end remove

END PROGRAM testignore
```

Transformed code

```
PROGRAM testignore

  !$claw remove
  PRINT*,'These lines'
  PRINT*,'are ignored'
  PRINT*,'by the CLAW compiler'
  PRINT*,'but kept in the final transformed code'
  PRINT*,'with the remove directives.'
  !$claw end remove

END PROGRAM testignore
```

5.3 Verbatim

```
!$claw verbatim IF (i > 5) THEN

  ! code block

!$claw verbatim END IF
```

The `verbatim` directive allows the user to insert code after the transformation process. This code is not analyzed by the compiler.

Code example

In the example below, the first remove block is ignored and the second is not.

Original code

```
PROGRAM testverbatim

```

```
  !$claw verbatim IF (.FALSE.) THEN
  PRINT*,'These lines'
  PRINT*,'are not printed'
  PRINT*,'if the the CLAW compiler has processed'
  PRINT*,'the file.'
  !$claw verbatim END IF

END PROGRAM testverbatim
```

Transformed code

```
PROGRAM testverbatim

IF (.FALSE.) THEN
PRINT*,'These lines'
PRINT*,'are not printed'
PRINT*,'if the the CLAW compiler has processed'
PRINT*,'the file.'
END IF

END PROGRAM testverbatim
```

A Code examples

A.1 kcache 2

Assignment is created during the transformation.

Original code

```
DO j3 = ki3sc+1, ki3ec
  !$claw kcache data(array2)
  var1 = x * y - array2(j1, j3)
  var2 = z * array2(j1, j3)
END DO
```

Transformed code

```
DO j3 = ki3sc+1, ki3ec
  array2_k = array2(j1,j3)
  var1 = x * y - array2_k
  var2 = z * array2_k
END DO
```

A.2 kcache 3

Using `init` and `private` clause

Original code

```
!$acc parallel
DO j3 = ki3sc+1, ki3ec
  var3 = array1(j1,j3-1)

  !$claw kcache data(array1) offset(0 -1) init private
  array1(j1,ki3sc) = x * y * z
  var1 = x * y - array1(j1, j3-1)
  var2 = z * array1(j1, j3-1)
END DO
!$acc end parallel
```

Transformed code

```
!$acc parallel private(array1_k_m1)
DO j3 = ki3sc+1, ki3ec
  IF (j3 == ki3sc+1) THEN
    array1_k_m1 = array1(j1,j3-1)
  END IF
  var3 = array1_k_m1
  array1_k_m1 = x * y * z
  array1(j1,ki3sc) = array1_k_m1
  var1 = x * y - array1_k_m1
  var2 = z * array1_k_m1
END DO
!$acc end parallel
```

A.3 loop-interchange 2

Example with 3 levels of loop.

Original code

```
!$claw loop-interchange (k,i,j)
DO i=1, iend     ! loop at depth 0
  DO j=1, jend   ! loop at depth 1
    DO k=1, kend ! loop at depth 2
      ! loop body here
    END DO
  END DO
END DO
```

Transformed code

```
! CLAW transformation (loop-interchange (k,i,j))
DO k=1, kend       ! loop at depth 2
  DO i=1, iend     ! loop at depth 0
    DO j=1, jend   ! loop at depth 1
      ! loop body here
    END DO
  END DO
END DO
```

A.4 loop-interchange 3

Example with OpenACC directives.

Original code

```
!$acc parallel
!$acc loop gang
!$claw loop-interchange
DO i=1, iend
  !$acc loop vector
  DO k=1, kend
    ! loop body here
  END DO
END DO
!$acc end parallel
```

Transformed code

```
! CLAW transformation (loop-interchange i < -- > k)
!$acc parallel
!$acc loop gang
DO k=1, kend
  !$acc loop vector
  DO i=1, iend
    ! loop body here
  END DO
END DO
!$acc end parallel
```

A.5 loop-fusion 2

Example using the *group* clause.

Original code

```
DO k=1, iend
  !$claw loop-fusion group(g1)
  DO i=1, iend
    ! loop #1 body here
  END DO

  !$claw loop-fusion group(g1)
  DO i=1, iend
    ! loop #2 body here
  END DO

  !$claw loop-fusion group(g2)
  DO i=1, jend
    ! loop #3 body here
  END DO

  !$claw loop-fusion group(g2)
  DO i=1, jend
    ! loop #4 body here
  END DO
END DO
```

Transformed code

```
DO k=1, iend
  ! CLAW transformation (loop-fusion group g1)
  DO i=1, iend
    ! loop #1 body here
    ! loop #2 body here
  END DO

  ! CLAW tranformation (loop-fusion group g2)
  DO i=1, jend
    ! loop #3 body here
    ! loop #4 body here
  END DO
END DO
```

A.6 loop-fusion 3

Example showing behavior with other directives (OpenACC).

Original code

```
!$acc parallel
!$acc loop gang
DO k=1, iend
```

```
  !$acc loop seq
  !$claw loop-fusion
  DO i=1, iend
    ! loop #1 body here
  END DO

  !$acc loop vector
  !$claw loop-fusion
  DO i=1, iend
    ! loop #2 body here
  END DO
END DO
!$acc end parallel
```

Transformed code

```
!$acc parallel
!$acc loop gang
DO k=1, iend
  ! CLAW transformation (loop-fusion same block group)
  !$acc loop seq
  DO i=1, iend
    ! loop #1 body here
    ! loop #2 body here
  END DO
END DO
!$acc end parallel
```

A.7 loop-fusion 4

Example using the *collapse* clause.

Original code

```
DO k=1, iend
  !$claw loop-fusion collapse(2)
  DO i=0, iend
    DO j=0, jend
      ! nested loop #1 body here
    END FO
  END DO

  !$claw loop-fusion collapse(2)
  DO i=0, iend
    DO j=0, jend
      ! loop #2 body here
    END DO
  END DO
```

```
END DO
```

Transformed code

```
DO k=1, iend
  ! CLAW transformation (loop-fusion collapse(2))
  DO i=0, iend
    DO j=0, jend
      ! nested loop #1 body here
      ! nested loop #2 body here
    END DO
  END DO
END DO
```

A.8 loop-extract 2

Example with *fusion* clause.

Original code

```
PROGRAM main
  !$claw loop-extract(i=istart,iend) map(value1,value2:i) fusion group(g1)
  CALL xyz(value1, value2)

  !$claw loop-fusion group(g1)
  DO i = istart, iend
    ! some computation here
    print*,'Inside loop', i
  END DO
END PROGRAM main

SUBROUTINE xyz(value1, value2)
  REAL, INTENT (IN) :: value2(x:y), value2(x:y)

  DO i = istart, iend
    ! some computation with value1(i) and value2(i) here
  END DO
END SUBROUTINE xyz
```

Transformed code

```
PROGRAM main
  !CLAW extracted loop
  DO i = istart, iend
    CALL xyz_claw(value1(i), value2(i))
    ! some computation here
    print*,'Inside loop', i
```

```
  END DO
END PROGRAM main

SUBROUTINE xyz(value1, value2)
  REAL, INTENT (IN) :: value2(x:y), value2(x:y)

  DO i = istart, iend
    ! some computation with value1(i) and value2(i) here
  END DO
END SUBROUTINE xyz

!CLAW extracted loop new subroutine
SUBROUTINE xyz_claw(value1, value2)
  REAL, INTENT (IN) :: value1, value2
  ! some computation with value1 and value2 here
END SUBROUTINE xyz_claw
```

A.9 loop-extract 3

Example with more complicated mapping.

Original code

```
PROGRAM main
  !$claw loop-extract(i=istart,iend) map(value1,value2:i/j)
  CALL xyz(value1, value2)
END PROGRAM main

SUBROUTINE xyz(value1, value2, j)
  INTGER, INTENT(IN) :: j
  REAL  , INTENT(IN) :: value2(x:y), value2(x:y)

  DO i = istart, iend
    ! some computation with value1(j) and value2(j) here
  END DO
END SUBROUTINE xyz
```

Transformed code

```
PROGRAM main
  !CLAW extracted loop
  DO i = istart, iend
    CALL xyz_claw(value1(i), value2(i))
  END DO
END PROGRAM main

SUBROUTINE xyz(value1, value2, j)
  INTGER, INTENT(IN) :: j
```

```
  REAL  , INTENT(IN) :: value2(x:y), value2(x:y)

  DO i = istart, iend
    ! some computation with value1(j) and value2(j) here
  END DO
END SUBROUTINE xyz

!CLAW extracted loop new subroutine
SUBROUTINE xyz_claw(value1, value2, j)
  INTGER, INTENT(IN) :: j
  REAL, INTENT (IN) :: value1, value2
  ! some computation with value1 and value2 here
END SUBROUTINE xyz_claw
```

A.10 array-transform 2

Example with *induction* and *acc* clauses

Original code

```
SUBROUTINE vector_add
  INTEGER :: i = 10
  INTEGER, DIMENSION(0:9) :: vec1

  !$acc parallel
  !$claw array-transform induction(myinduc) acc(loop)
  vec1(0:i) = vec1(0:i) + 10;

  !$claw array-transform acc(loop)
  vec1(0:i) = vec1(0:i) + 1;
  !$acc end parallel
END SUBROUTINE vector_add
```

Transformed code

```
SUBROUTINE vector_add
  INTEGER :: claw_i
  INTEGER :: myinduc
  INTEGER :: i = 10
  INTEGER, DIMENSION(0:9) :: vec1

  !$acc parallel

  ! CLAW transformation array notation vec1(0:i) to do loop
  !$acc loop
  DO myinduc = 0, i
    vec1(myinduc) = vec1(myinduc) + 10;
  END DO
```

```

  ! CLAW transformation array notation vec1(0:i) to do loop
  !$acc loop
  DO claw_i = 0, i
    vec1(claw_i) = vec1(claw_i) + 1;
  END DO

  !$acc end parallel
END SUBROUTINE vector_add
```

A.11 array-transform 3

Example with fusion clause

Original code

```
SUBROUTINE vector_add
  INTEGER :: i = 10
  INTEGER, DIMENSION(0:9) :: vec1
  INTEGER, DIMENSION(0:9) :: vec2

  !$claw array-transform induction(claw_i) fusion
  vec1(0:i) = vec1(0:i) + 10;

  !$claw array-transform induction(claw_i) fusion
  vec2(0:i) = vec2(0:i) + 1;
END SUBROUTINE vector_add
```

Transformed code

```
SUBROUTINE vector_add
  INTEGER :: claw_i
  INTEGER :: i = 10
  INTEGER, DIMENSION(0:9) :: vec1
  INTEGER, DIMENSION(0:9) :: vec2

  ! CLAW transformation array notation vec1(0:i) to do loop
  ! CLAW transformation array notation vec2(0:i) to do loop
  ! CLAW transformation fusion
  DO claw_i=0, i
    vec1(claw_i) = vec1(claw_i) + 10;
    vec2(claw_i) = vec2(claw_i) + 1;
  END DO
END SUBROUTINE vector_add
```

A.12 array-transform 4

Example with 2-dimensional arrays

Original code

```
SUBROUTINE vector_add
INTEGER :: i = 10
INTEGER, DIMENSION(0:10,0:10) :: vec1
INTEGER, DIMENSION(0:10,0:10) :: vec2

vec1(0:i,0:i) = 0;
vec2(0:i,0:i) = 100;

!$claw array-transform
vec1(0:i,0:i) = vec2(0:i,0:i) + 10
END SUBROUTINE vector_add
```

Transformed code

```
SUBROUTINE vector_add
INTEGER :: i = 10
INTEGER, DIMENSION(0:10,0:10) :: vec1
INTEGER, DIMENSION(0:10,0:10) :: vec2

vec1(0:i,0:i) = 0;
vec2(0:i,0:i) = 100;

DO claw_i = 0, i, 1
  DO claw_j = 0, i, 1
    vec1(claw_i,claw_j) = vec2(claw_i, claw_j) + 10
  END DO
END DO
END SUBROUTINE vector_add
```

A.13 remove 2

Original code

```
DO k=1, kend
  DO i=1, iend
    ! loop #1 body here
  END DO

  !$claw remove
  PRINT*, k
  PRINT*, k+1
```

```
  !$claw end remove

  DO i=1, iend
    ! loop #2 body here
  END DO
END DO
```

Transformed code

```
DO k=1, kend
  DO i=1, iend
    ! loop #1 body here
  END DO

  DO i=1, iend
    ! loop #2 body here
  END DO
END DO
```