International Fortran Conference 2020

Generic Programming Techniques

by the example of tensor contraction

Patrick Seewald

International Fortran Conference, University of Zurich, July 3, 2020

Department of Chemistry, University of Zurich

Background

- PhD student in theoretical chemistry
- **CP2K** project: quantum chemistry & solid state physics, Fortran 2008
  `https://github.com/cp2k/cp2k`
- CP2K is strong in algorithms based on sparse linear algebra using the sparse matrix/tensor library **DBCSR**
  `https://github.com/cp2k/dbcsr`
- Fortran tools used in CP2K / DBCSR:
  - **Fypp**: Preprocessor (Python-based templates / macros)
    `https://github.com/aradi/fypp`
  - **fprettify**: Auto formatter (whitespace / indentation)
    `https://github.com/pseewald/fprettify`
  - **FORD**: Documentation generator
    `https://github.com/Fortran-FOSS-Programmers/ford`
- My work: algorithms based on sparse tensor contractions, generalization of DBCSR sparse matrix library to tensors

Generic Programming in General

- application of the same algorithm to multiple data types

$\rightarrow$ e.g. sort implemenation for arbitrary data types

- solve a class of related problems instead of tackling each specific problem on its own

$\rightarrow$ general algorithm that can be applied to many different problems

Generic Programming with Fortran

Important generic programming ingredients:

- **Polymorphism** (run-time): generic type representing multiple specific types (e.g. a common shape class for rectangles and triangles)
  ☑ Fortran 2003
- **Templates and macros** (compile-time): generate code for different types
  ☐ Fortran standard does not include templates/macros. Compilers implement a basic preprocessor (cpp/fpp) restricted to including common code snippet.

Typical workarounds for missing Fortran macros/templates:

- **Code duplication**: limited and problematic (bug fixes, refactoring)
- **Code generators**: delegate generic programming to another language
- **Non-standard preprocessors**: extend Fortran syntax with an external macro language

Example: tensor contractions / Einstein summation

$$\sum_k A_{ijk} B_{kj} = C_i \qquad \sum_{i,j} A_{ijkl} B_{jim} = C_{mkl}$$

Generic Fortran API:

```fortran
class(tensor), allocatable :: a, b, c

real, dimension(:,:,:), allocatable :: data_a
real, dimension(:,:), allocatable :: data_b

! ... allocate and assign data_a, data_b ...

a = tensor(data_a)
b = tensor(data_b)

c = tensor_einsum( &
   a, [1,2,3], b, [3,2], [1])
```

```fortran
class(tensor), allocatable :: a, b, c

integer, dimension(:,:,:,:), allocatable :: data_a
integer, dimension(:,:,:), allocatable :: data_b

! ... allocate and assign data_a, data_b ...

a = tensor(data_a)
b = tensor(data_b)

c = tensor_einsum( &
   a, [1,2,3,4], b, [2,1,5], [5,3,4])
```

Fortran API as simple as commonly used Python libraries (Numpy, PyTorch)

Implementation in 500 lines of Fortran:

https://github.com/pseewald/fortran-einsum-example

Example: tensor contractions / Einstein summation

$$\sum_{k} A_{ijk} B_{kj} = C_i \qquad \sum_{i,j} A_{ijkl} B_{jim} = C_{mkl} \qquad \sum_{i,j,k} A_{ijk} B_{jik} = C$$

Naive / direct implementation implementation:

```
DO i=1,SIZE(A,1)
DO j=1,SIZE(A,2)
DO k=1,SIZE(A,3)
   C(i) = C(i) + &
      A(i,j,k)*B(k,j)
ENDDO
ENDDO
ENDDO
```

```
DO i=1,SIZE(A,1)
DO j=1,SIZE(A,2)
DO k=1,SIZE(A,3)
DO l=1,SIZE(A,4)
DO m=1,SIZE(B,3)
   C(m,k,l) = C(m,k,l) + &
      A(i,j,k,l)*B(j,i,m)
ENDDO
ENDDO
ENDDO
ENDDO
ENDDO
```

```
DO i=1,SIZE(A,1)
DO j=1,SIZE(A,2)
DO k=1,SIZE(A,3)
   C = C + &
      A(i,j,k)*B(j,i,k)
ENDDO
ENDDO
ENDDO
```

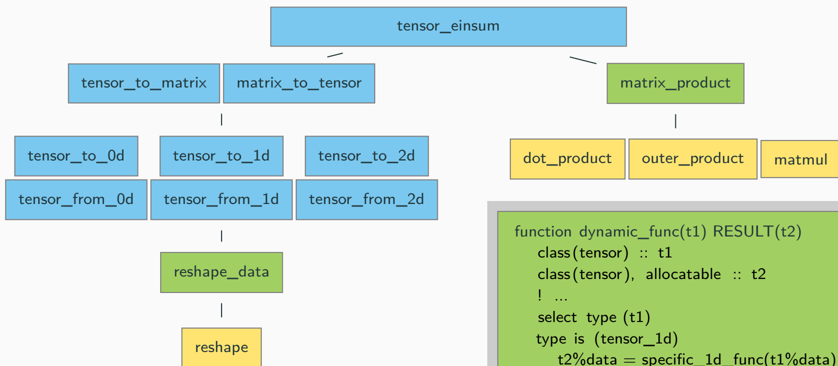
Generated code also needs to be optimized (loop unrolling, blocking, parallelization) $\rightarrow$ not a good starting point

Generic einsum implementation: strategy

- all tensor contractions can be mapped to products of matrices and vectors $\rightarrow$ reuse of existing libraries (e.g. Fortran intrinsics or BLAS) for all floating point operations:
  - $\sum_k A_{ijk} B_{kj} = C_i$ matrix-vector product
  - $\sum_{i,j} A_{ijkl} B_{jim} = C_{mkl}$ matrix-matrix product
  - $\sum_{i,j,k} A_{ijk} B_{jik} = C$ vector-vector inner product
  - $A_{ij} B_k = C_{ijk}$ vector-vector outer product
- tensor type should incorporate
  - different data types (integer/real/complex in 4-byte/8-byte precision)
  - different tensor ranks (0–7)

  $\rightarrow$ $6 \cdot 8 = 48$ different base data types, need a code generator or preprocessor
- generic implementation based on following hierarchy:
  1. generic: generic algorithms working on generic types (abstract classes)
  2. dynamic: mapping generic algorithms to specific implementations (select type construct)
  3. specific: implementations for all specific types (code generation)

Generic code design




```
function dynamic_func(t1) RESULT(t2)
    class(tensor) :: t1
    class(tensor), allocatable :: t2
    ! ...
    select type (t1)
    type is (tensor_1d)
        t2%data = specific_1d_func(t1%data)
    type is (tensor_2d)
        t2%data = specific_2d_func(t1%data)
    ! ...
    end select
end function
```

```
function generic_func(t1) RESULT(t2)
    class(tensor) :: t1
    class(tensor), allocatable :: t2
    t2 = dynamic_func(t1)
    ! ...
end function
```

```
function specific_2d_func(t1) RESULT(t2)
    real, intent(in) :: t1(:, :)
    real, allocatable :: t2(:, :)
    ! ...
end function
```

Tensor einsum: types

```
! abstract data type
type, abstract :: tensor_data
end type

! concrete data types (48 instances)
type, extends(tensor_data) :: data_0d_i4
   integer(int32), allocatable :: d
end type
type, extends(tensor_data) :: data_0d_i8
   integer(int64), allocatable :: d
end type
type, extends(tensor_data) :: data_0d_r4
   real(real32), allocatable :: d
end type
! ...
type, extends(tensor_data) :: data_1d_i4
   integer(int32), allocatable :: d(:)
end type
! ...
type, extends(tensor_data) :: data_2d_i4
   integer(int32), allocatable :: d(:,:)
end type
```

Automated type generation using Fypp preprocessor:

```
#:for rank in ranks
#:for name, type in data_params
   type, extends(tensor_data) :: &
      data_${rank}$d_${name}$
      ${type}$, allocatable :: d${shape(rank)}$
   end type
#:endfor
#:endfor
```

Fypp – Python powered Fortran metaprogramming

https://github.com/aradi/fypp

- Iterated output to simulate templates

```
interface myfunc
#:for dtype in ['real', 'dreal', 'complex', 'dcomplex']
  module procedure myfunc_${dtype}$
#:endfor
end interface myfunc
```

- Macros

```
#:def ASSERT(cond)
  #:if DEBUG > 0
    if (.not. ${cond}$) then
      print *, "Assert failed in file ${_FILE_}$, line ${_LINE_}$"
      error stop
    end if
  #:endif
#:enddef ASSERT

@:ASSERT(size(myArray) > 0)
```

- Insertion of arbitrary Python expressions

```
character(*), parameter :: comp_date = "${time.strftime('%Y-%m-%d')}$"
```

Tensor einsum: macros

tensor_lib.fpp

```
#:set ranks = range(0, RANK+1)
#:set data_name = ['i4', 'i8', 'r4', 'r8', 'c4', 'c8']
#:set data_type = ['integer(int32)', 'integer(int64)', 'real(real32)', ...]
#:set data_params = list(zip(data_name, data_type))

#:def shape(n)
$:'' if n == 0 else '(' + ':' + ',:' * (n - 1) + ')'
#:enddef

! concrete data types generated using Fypp preprocessor
#:for rank in ranks
#:for name, type in data_params
   type, extends(tensor_data) :: data_${rank}$d_${name}$
      ${type}$, allocatable :: d${shape(rank)}$
   end type
#:endfor
#:endfor
```

fypp −DRANK=7 tensor_lib.fpp > tensor_lib.f90

```
! ...
type, extends(tensor_data) :: data_3d_i4
   integer(int32), allocatable :: d(:,:,:)
end type
! ...
```

Tensor einsum: types and constructor

```
! abstract tensor type
type, abstract :: tensor
   integer, dimension(:), allocatable :: shape
   class(tensor_data), allocatable :: data
end type

! concrete tensor types
#:for rank in ranks
   type, extends(tensor) :: tensor_${rank}$d
#:if rank == 1
      integer :: vector_type = row_vec
#:endif
   end type
#:endfor

! constructor
interface tensor
#:for rank in ranks
#:for name in data_name
   module procedure tensor_${rank}$d_${name}$
#:endfor
#:endfor
end interface
```

```
! constructor implementation
#:for rank in ranks
#:for name, type in data_params
   function tensor_${rank}$d_${name}$ (data) &
      result(t)

      ${type}$, intent(in) :: data${shape(rank)}$
      integer, dimension(${rank}$) :: sh
      type(tensor_${rank}$d), allocatable :: &
         t_${rank}$d
      class(tensor), allocatable :: t
      type(data_${rank}$d_${name}$), &
         allocatable :: t_data

#:if rank > 0
      sh = shape(data)
#:endif
      allocate (t_${rank}$d)
      allocate (t_${rank}$d%shape(${rank}$), &
               source=sh)

      allocate (t_data)
      allocate (t_data%d, source=data)
      call move_alloc(t_data, t_${rank}$d%data)

      call move_alloc(t_${rank}$d, t)
   end function
#:endfor
#:endfor
```

Tensor einsum: API procedure (generic)

```fortran
function tensor_einsum(tensor_1, ind_1, tensor_2, ind_2, ind_3) result(tensor_3)
   class(tensor), intent(in) :: tensor_1
   integer, dimension(:), intent(in) :: ind_1
   class(tensor), intent(in) :: tensor_2
   integer, dimension(:), intent(in) :: ind_2
   class(tensor), allocatable :: tensor_3
   integer, dimension(:), intent(in) :: ind_3
   integer, dimension(:), allocatable :: &
      ind_1_l, ind_1_r, ind_2_l, ind_2_r, ind_3_l, ind_3_r, t3_shape
   class(tensor), allocatable :: matrix_1, matrix_2, matrix_3
   integer :: i

   call index_einstein_to_matrix_product( &
      ind_1, ind_2, ind_3, ind_1_l, ind_1_r, ind_2_l, ind_2_r, ind_3_l, ind_3_r)

   matrix_1 = tensor_to_matrix(tensor_1, ind_1_l, ind_1_r)
   matrix_2 = tensor_to_matrix(tensor_2, ind_2_l, ind_2_r)

   matrix_3 = matrix_product(matrix_1, matrix_2)

   allocate (t3_shape(size(ind_3_l) + size(ind_3_r)))
   t3_shape([ind_3_l, ind_3_r]) = [tensor_1%shape(ind_1_l), tensor_2%shape(ind_2_r)]

   tensor_3 = tensor_from_matrix(matrix_3, t3_shape, ind_3_l, ind_3_r)

end function
```

Tensor einsum: matrix product (dynamic)

```
   function matrix_product(matrix_1, matrix_2) result(matrix_3)
      class(tensor), intent(in) :: matrix_1, matrix_2 ! dynamic type tensor_1d or tensor_2d
      class(tensor), allocatable :: matrix_3 ! dynamic type tensor_0d, tensor_1d or tensor_2d

      select type (matrix_1)
      type is (tensor_1d)
         select type (matrix_2)
         type is (tensor_1d)
            if (matrix_1%vector_type == row_vec .and. matrix_2%vector_type == col_vec) then
               select type (data_1 => matrix_1%data)
#:for name in data_name
               type is (data_1d_${name}$)
                  select type (data_2 => matrix_2%data)
                  type is (data_1d_${name}$)
                     matrix_3 = tensor(dot_product(data_1%d, data_2%d))
                  end select
#:endfor
               end select
            elseif (matrix_1%vector_type == col_vec .and. matrix_2%vector_type == row_vec) then
            ! ...
               matrix_3 = tensor(outer_product(data_1%d, data_2%d))
            ! ...
            endif
         type is (tensor_2d)
            ! ...
            matrix_3 = tensor(matmul(data_1%d, data_2%d))
            ! ...
         end select
      type is (tensor_2d)
         ! ... and so on ...
```

Tensor einsum: vector outer product (specific)

matmul and dot_product are Fortran intrinsics but we need to implement outer_product:

```
interface outer_product
#:for name in data_name
   module procedure outer_product_${name}$
#:endfor
end interface

#:for name, type, kind in data_params
function outer_product_${name}$ (vector_1, vector_2) result(matrix)
   ${type}$, dimension(:), intent(in) :: vector_1, vector_2
   integer :: k, l
   ${type}$, dimension(:, :), allocatable :: matrix

   allocate (matrix(size(vector_1), size(vector_2)))
   do k = 1, size(vector_1)
      do l = 1, size(vector_2)
         matrix(k, l) = vector_1(k)*vector_2(l)
      enddo
   enddo
end function
#:endfor
```

Conclusions

- **Generic programming** to implement complex problems in less lines of codes
- **Modern Fortran APIs** can be as elegant / simple as commonly used Python packages
- Two main ingredients to enable generic programming in Fortran:
  1. **Preprocessor** to automate generation of all type-specific code
  2. **OOP** and **polymorphism** to create generic types and methods instead of many type-specific instances
- **Fypp** preprocessor as powerful as custom code generators but easier and safer to use
- **Limitations**:
  - all templates are explicitly instantiated and compiled $\rightarrow$ large binary size and long compilation time.
  - Mixing Fortran with preprocessor language is hard to read and debug
- Example code: `https://github.com/pseewald/fortran-einsum-example`