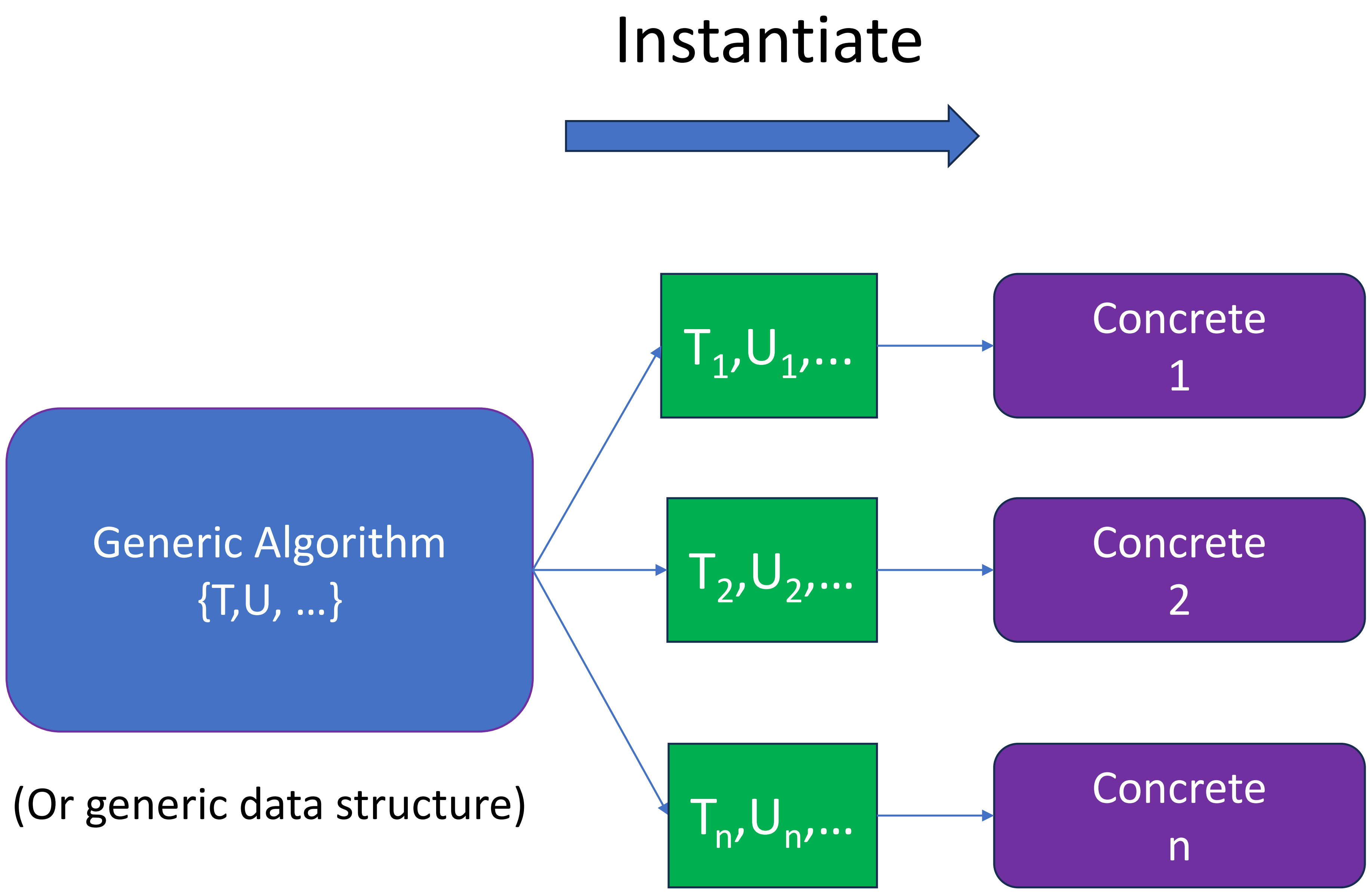
Type-safe Generic Programming in Fortran

Tom Clune
Global Modeling and Assimilation Office
NASA Goddard Space Flight Center

- Introduction
- Perspective
- Brief history of Fortran
- Existing features that support generic programming
- Proposed features and syntax
- Current status
- Future extensions

1. The release date of the next Fortran revision is not certain.   Probably 2028 or 2029
    - Internal name is “F202y”
2. The feature set for F202y is not yet frozen (but nearly so)
3. In particular, the syntax for generic programming is still going through approval process
4. Suggestions for changes are still welcome, but ...
    - Any nontrivial changes may prevent inclusion of generic programming in F202y
5. Interesting examples of generic algorithms are generally a bit large for PPT format.
    - Hand-waving, pseudocode and trivial examples must suffice.



Instantiate

Generic Algorithm {T,U, …}

$T_1,U_1,\ldots$

$T_2,U_2,\ldots$

$T_n,U_n,\ldots$

Concrete 1

Concrete 2

Concrete n

(Or generic data structure)

Allows reusability and efficiency

A ***generic algorithm*** (or data structure) is parameterized by <u>types</u> that are *deferred* (specified later/elsewhere).

- For Fortran we extend this to "*… parameterized by* ***types***, ***kinds,*** *or* ***ranks*** *that are deferred …*"
- Most approaches also allow for parameterization by *operations (i.e., procedures)* that are deferred.

Instantiation associates actual types for the deferred parameters to produce *concrete* implementations.

Multidimensional Arrays

- Data structure that holds elements of some type (special case of “container”)
- Optimized interface for random access

Supported generic algorithms on arrays:

- Reference/assignment to individual element or slice
- Allocation/deallocation
- Elemental operations
- Various intrinsics: SIZE(), SHAPE(), RANK(), RESHAPE(), PACK(), SPREAD(), …

“Rank-agnostic” features (new in F2023)

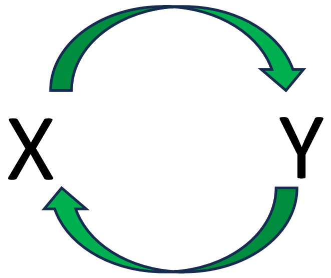
- Permit writing algorithm that is parameterized by rank
- Declaraton
  - REAL, BOUNDS(0*SHAPE(X):SHAPE(X)+1) :: x_with_guard_cells
  - REAL, ALLOCATABLE, RANK(N) :: tmp ! N is integer constant
- Allocation
  - ALLOCATE(tmp(v1:v2)) ! v1 and v2 are1D integer arrays of the same size
- Element/slices
  - A(**@**[1,5]) = B(**@**v) ! v is a 1D integer array of size 2
  - C(**@**v1,:,**@**v2) ! V1, v2 are 1D integer arrays

Miscellaneous:

- Various intrinsics restricted to numeric types: MINLOC, MAXLOC, SUM, …
- Parameterized derived types (parameterized by deferred “type-parameters”)
- INCLUDE statement
- Intrinsics
  - STORAGE_SIZE(), MERGE(), TRANSFER(), …
- SELECT RANK(…) (F2018)
- TYPEOF(), CLASSOF() (F2023)

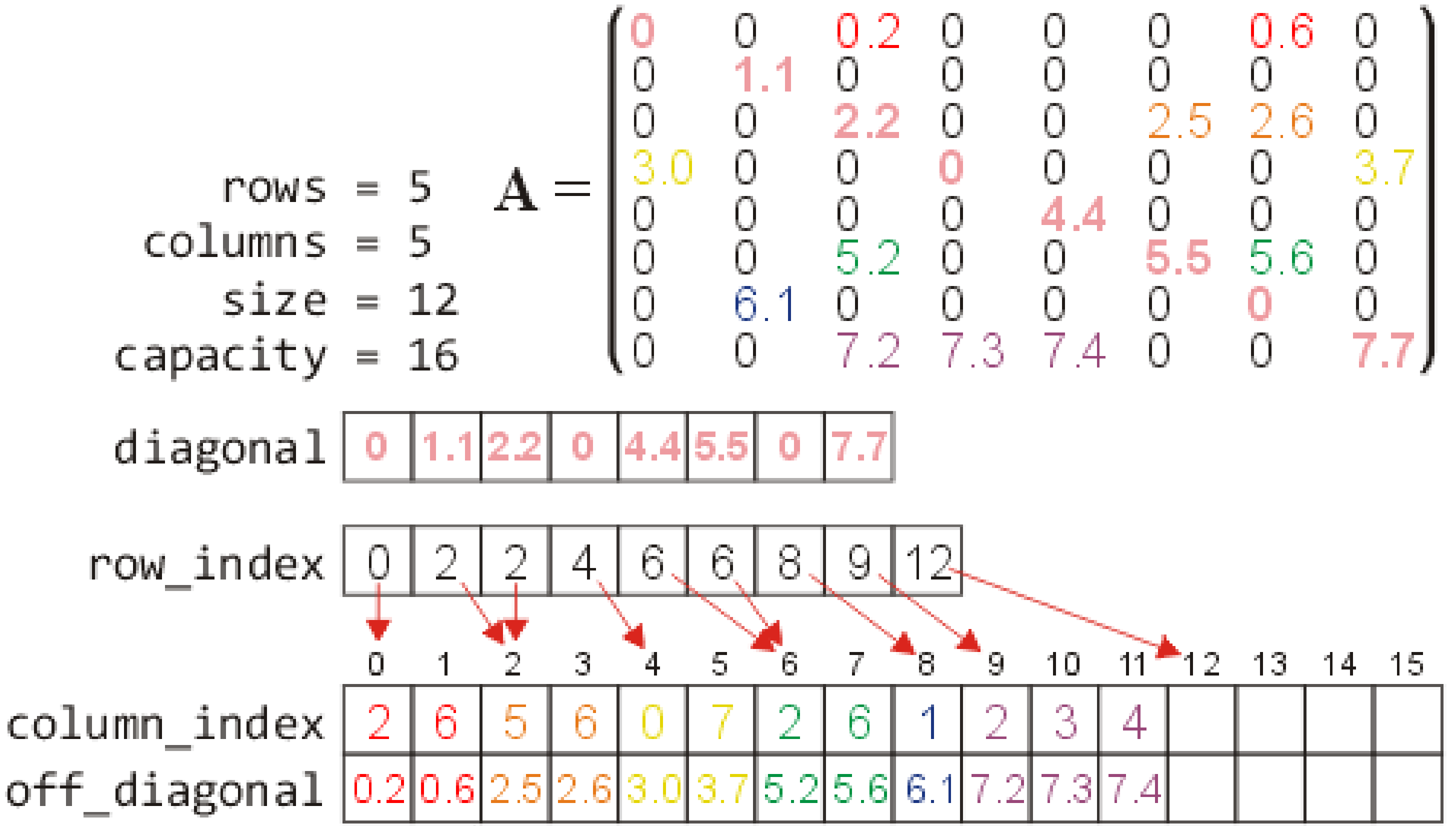
- Generic procedures (expected in F202y)
  - Automatic overload for a *specified* set of types & kinds (<u>not deferred</u>)

- **Swap**: Provide a procedure that can swap the contents of 2 variables of deferred type T



- **Generalized intrinsic procedures**: E.g., FINDLOC()
  - Most require an additional operation (e.g., “==”)

- **Containers**: Objects that store a collection of objects of deferred type T
  - Specialized/optimized accessors
  - Examples
    - Fortran arrays
    - vector, stack, queue,…
    - set, map, tree
    - sparse arrays



```
     rows = 5
  columns = 5
     size = 12
 capacity = 16
```

$$\mathbf{A} = \begin{pmatrix} 0 & 0 & 0.2 & 0 & 0 & 0 & 0.6 & 0 \\ 0 & 1.1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2.2 & 0 & 0 & 2.5 & 2.6 & 0 \\ 3.0 & 0 & 0 & 0 & 0 & 0 & 0 & 3.7 \\ 0 & 0 & 0 & 0 & 4.4 & 0 & 0 & 0 \\ 0 & 0 & 5.2 & 0 & 0 & 5.5 & 5.6 & 0 \\ 0 & 6.1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 7.2 & 7.3 & 7.4 & 0 & 0 & 7.7 \end{pmatrix}$$

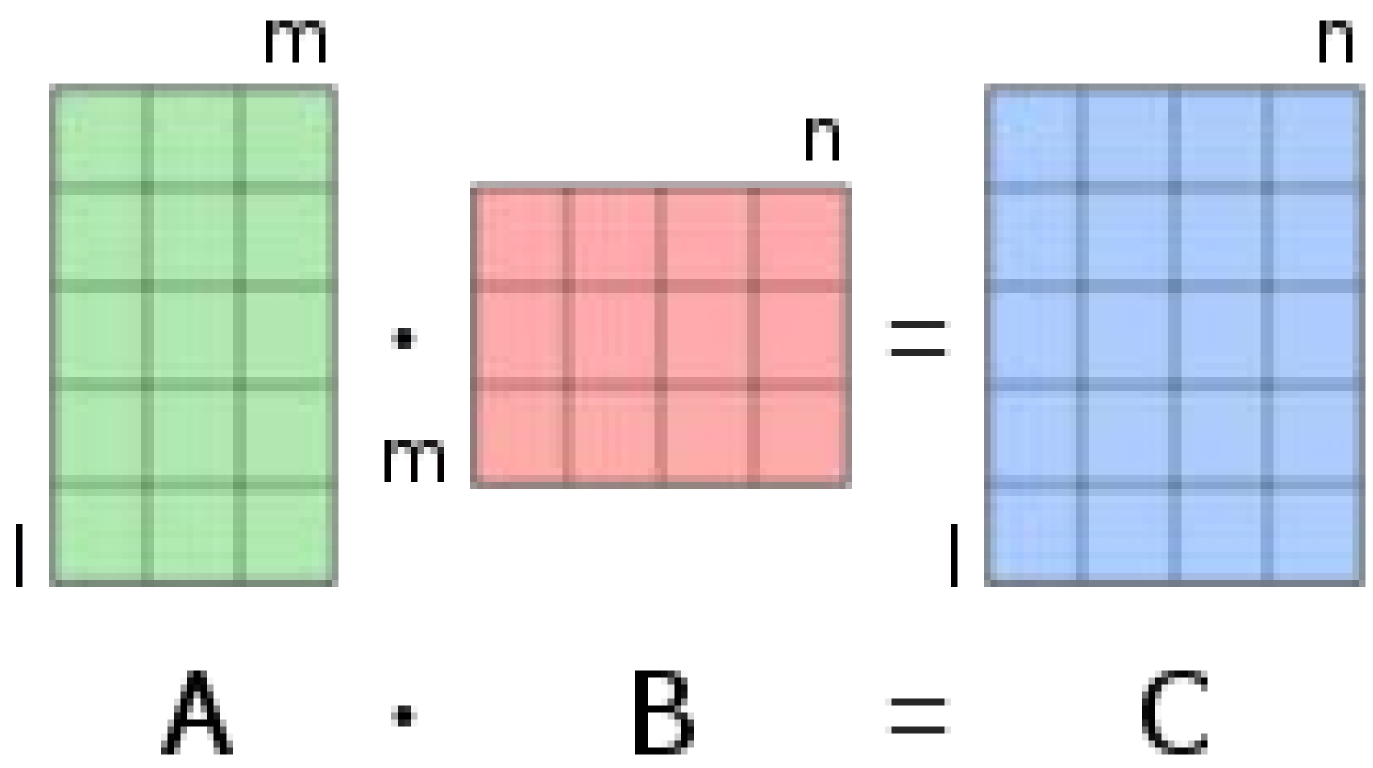
diagonal	0	1.1	2.2	0	4.4	5.5	0	7.7

row_index	0	2	2	4	6	6	8	9	12

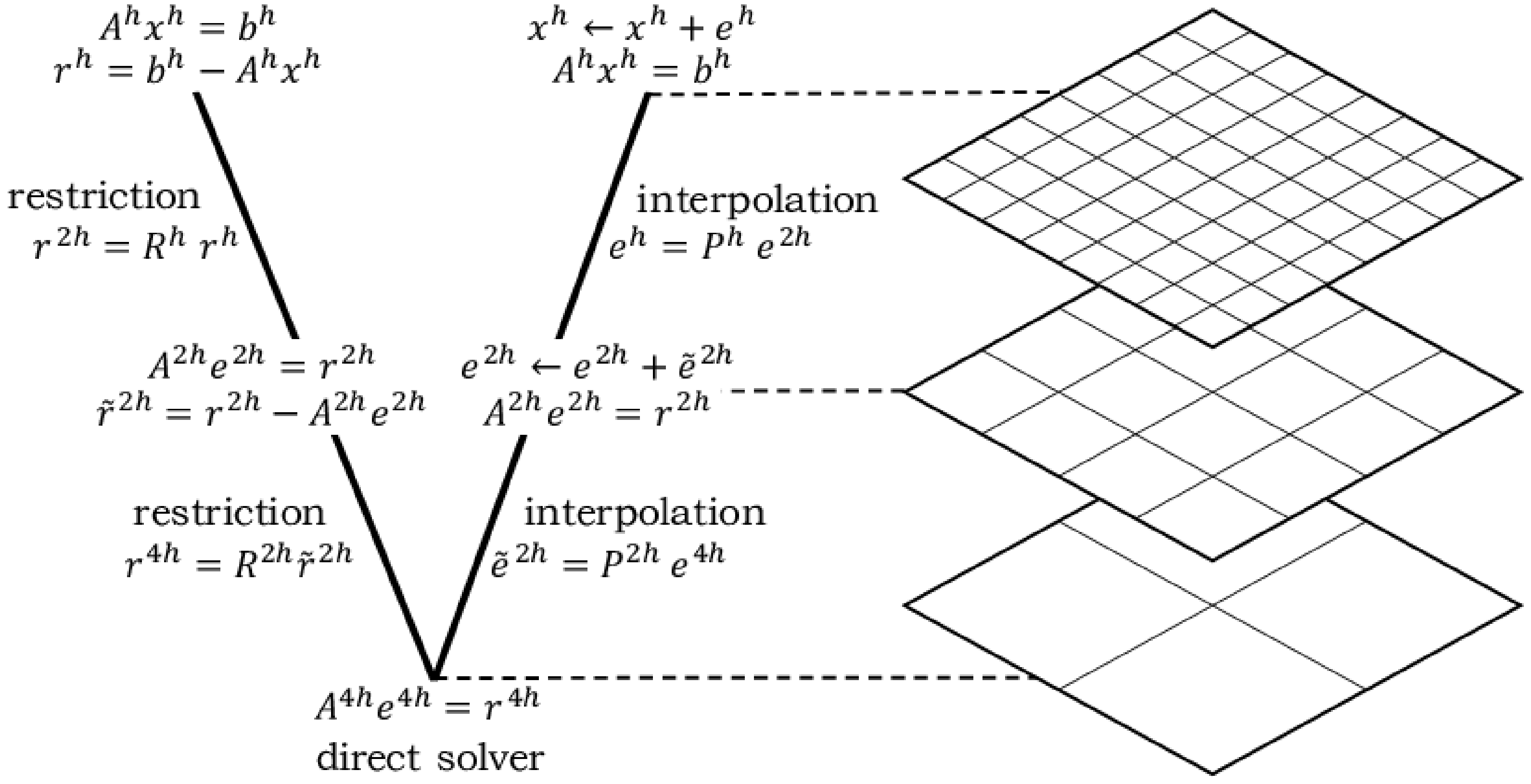
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
column_index	2	6	5	6	0	7	2	6	1	2	3	4				
off_diagonal	0.2	0.6	2.5	2.6	3.0	3.7	5.2	5.6	6.1	7.2	7.3	7.4				

Linear algebra:

- Operations
  - Matrix-matrix multiplication
  - Solution of linear systems
  - …
- Reuse for
  - Block-matrices
  - Sparse arrays
  - Tropical semirings (”+”=min, “*”=+)



- Multigrid methods



Ibeid, Huda & Olson, Luke & Gropp, William. (2018). FFT, FMM, and Multigrid on the Road to Exascale: performance challenges and opportunities.

Magne Haveraaen

Brad Richardson

Damian Rouson

Ondrej Certik

General approach by subgroup

- Strong reliance on committee-approved use cases
- Deference to advice from outside consultant (M. Haveraaen)
- Conservative: much easier to *relax* a constraint in the future than to *add* one later.

Goal: *Enable simple implementation of generics while avoiding problems found in other languages.*

1. **A generics facility needs to be typesafe, for both developers and users of generic code.** (C++)
    - Library developers: type system rejects code that relies on features not specified in the generic interface
    - Library users: type system rejects instantiation with parameters that do not satisfy the interface.
    - *Early detection of errors: definition vs instantiation vs link time*
    - *Avoid obscure error messages*

2. **Both *intrinsic* and *user-defined* types and procedures need to be treated uniformly**
    - Any disparity will cause problems for code reuse down the line. (JAVA)
    - Generic mechanism should allow operations as parameters

3. **A generics facility needs to allow generic parameter lists to be *extended/nested*.**
    - Some constructs will require additional parameters.
    - E.g., Operations on sparse matrices require additional operations:
        - Minimum element: MIN
        - Sum of all elements: OPERATOR(+)

TEMPLATE A{T}

```
INSTANTIATE B{T, REAL}
```

TEMPLATE B{T,U}

```
TYPE(T) :: x
TYPE(U) :: y
…
y = y + FLOOR(x)
```

Library developer tests with:

```
INSTANTIATE A{REAL} ! OK

INSTANTIATE A{REAL(KIND(1.D0))} ! OK
```

Clever user comes along:

```
INSTANTIATE A{COMPLEX} ! INVALID
```

Error: 'a' argument of 'floor' intrinsic at (1) must be REAL

User: *“What?”*

TEMPLATE A{T}

```
INSTANTIATE B{T, REAL}
```

TEMPLATE B{T,U}

```
TYPE(T) :: x
TYPE(U) :: y
…
CALL reduce(x, y) ! implicit
```

Library developer tests with:

```
INSTANTIATE A{REAL} ! OK

INSTANTIATE A{REAL(KIND(1.D0))} ! OK
```

Clever user comes along:

```
INSTANTIATE A{INTEGER} ! INVALID
```

```
Undefined symbols for architecture x86_64:
  "_reduce_", referenced from:
     _MAIN__ in ccTaPCui.o
ld: symbol(s) not found for architecture x86_64
collect2: error: ld returned 1 exit status
```

Consider a package that aims to deliver a variety of generic data structures and algorithms to support linear algebra applications:

Matrix multiplication: $C_{ij} = C_{ij} + \sum_k A_{ik} * B_{kj}$

- 4 type parameters
  - T (type of A)
  - U (type of B)
  - V (type of C)
  - W (type of A*B)
- 2 operations
  - Multiply (T * U → W)
  - Add (V + W → V)

Linear solvers: $A\,x = b$

- Needs 6 parameters above and …
- 2 additional operations:
  - Divide (T / T → W)
  - Minus (V - W → V)

Some users may only want matrix multiplication.

Design options:

1. Force such users to invent “-” and “/” operations
2. Provide two separate templates
   - Combinatorial explosion for realistic package
3. ***Provide solvers as an inner template***
   - Only users wanting to use inner template provide additional operations

Summary: Type and/or kind of procedure dummy argument is inferred from call site.

```
TEMPLATE SUBROUTINE s(x, y, r, z)
  TYPE(*) :: x, y, z
  REAL(*):: r
  z = x * y – y*r
END
. . .
COMPLEX :: a, c
INTEGER :: i
REAL(kind=kind(1.d0)) :: d
CALL s(a, i, d, c) ! Instantiates s
```

- **Pros**:
  - Relatively easy to specify and implement
  - Succinct
- **Cons**:
  - Not type-safe (bad for library developers and users)
  - Does not support generic data structures (e.g., containers)
  - Does not support more complex use cases with “intermediate types”
  - Likely obscure error messages

Challenges for intrinsic types

- Not extensible; e.g., `CLASS(INTEGER)`
- A few special cases cannot appear in ”variable definition context”; e.g., `EVENT_TYPE`
- Intrinsic operators on intrinsic types are not usually implemented as functions (inline)
  - Burden on implementors

Intrinsic procedures

- Some intrinsics are “magic” (technical term)
- Restrictions on overloading intrinsic operators on intrinsic types.

Summary: Powerful language-aware preprocessor

```
DEFINE MACRO :: iterator(count,operation)
  MACRO DO i=1,count
    EXPAND operation(i)
  MACRO END DO
END MACRO
DEFINE MACRO :: process_element(j)
  READ *,a(j)
  result(j) = process(a(j))
  IF (j>1) PRINT *,’difference =’,result(j)-result(j-1)
END MACRO
EXPAND iterator(17,process_element)! expands into 17 sets of 3 statements
```

- **Pros**:
  - Significant prior work; ready-to-roll
  - Supports generic algorithms and generic data structures
  - Supports *compile-time* conditionals and loops
  - Recursive
- **Cons**:
  - Yet another preprocessor …
  - Type-checking only at instantiation
  - Recursive

Summary: Overload a procedure for a specified set of types and kinds.

Technically not generic programming

```
GENERIC FUNCTION has_nan(x) RESULT(ans)
  REAL(REAL32,REAL64,REAL128), RANK(0), INTENT(IN) :: x
  LOGICAL :: ans

  ans = ieee_is_nan(x)

END FUNCTION has_nan
```

- **Pros**:
  - Supports a very common use case: *Overload legacy library for multiple real kinds.*
  - Complementary to templates
- **Cons**:
  - Overload types must be specified in advance

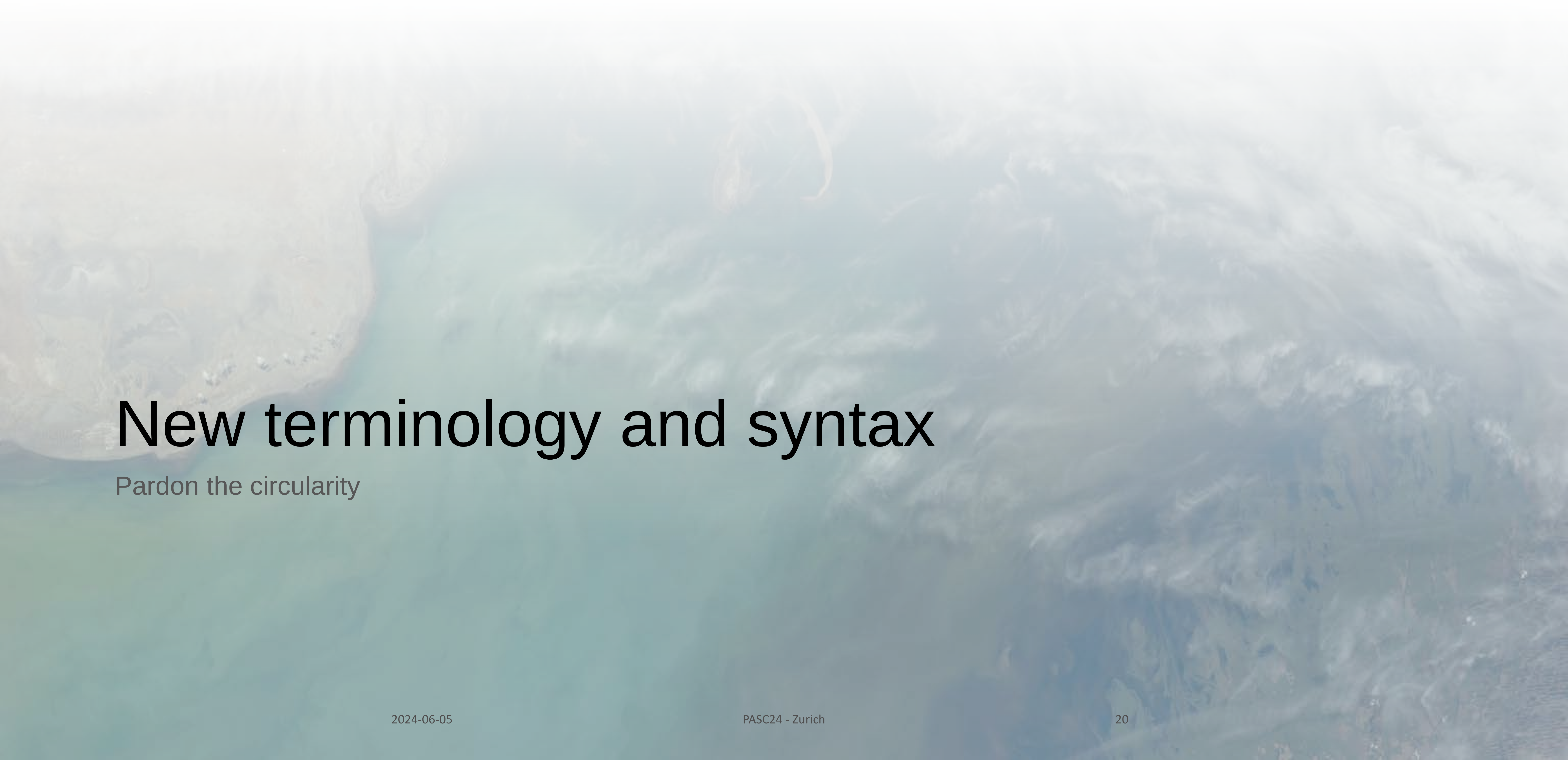
Summary: Type-safe built-in language construct which can contain generic algorithms and/or data structures

Pros:

- Implementation can be verified independently of instantiation
- Enables high-quality (clear) error messages for invalid instantiations.

• **Cons**:

- Nontrivial specification
- Nontrivial implementation (compiler)



New terminology and syntax

Pardon the circularity

Deferred arguments are used to parameterize templates (and requirements)

- Analogs of dummy arguments for procedures
- Associated with *instantiation arguments* specified in an INSTANTIATE statement

Supported forms:

- *Deferred types*:

  ```
  TYPE, DEFERRED :: T
  ```

- *Deferred constants*: Logical, Integer, or assumed-length character

  ```
  LOGICAL, CONSTANT :: FLAG
  INTEGER, CONSTANT :: COEFFICIENTS(:,:), TENSOR(..)
  ```

- *Deferred procedures*:
  - Interface can be in terms of other deferred arguments
  - Must have explicit interface
  - No new syntax for declaration

TEMPLATE: “module-like” construct that is parameterized by deferred arguments

- Template elements:
  - Name and list of deferred arguments
  - Specification part
  - Subprogram part (i.e, CONTAINS section)

- Important constraints and caveats
  - A template can only reference intrinsic procedures, procedures with explicit interfaces, and operators.
    - A.K.A. “strong concepts”
    - Note that *intrinsic assignment* is always permitted
  - Many Fortran constructs are not permitted within a template
    - Mostly because they would not make sense

- Templates can be nested
  - An “inner” template can access the deferred arguments of the outer component
  - Very useful when only some subset of a package requires additional parameters
  - E.g., Mat-Mat multiply requires procedures for addition and multiplication. But matrix solve requires additional operators for subtraction and division.

```
TEMPLATE swap_tmpl{T}
  TYPE, DEFERRED :: T
CONTAINS

  SUBROUTINE swap(x, y)
    TYPE(T), INTENT(INOUT) :: x, y
    TYPE(T) :: tmp

    tmp = x
    x = y
    y = x
  END SUBROUTINE

END TEMPLATE
```

- Templates can be nested
  - An “inner” template can access the deferred arguments of the outer component
  - Natural mechanism to extend a package with procedures that require additional parameters
- Example: Consider a linear algebra package that provides for abstract matrix-matrix multiplication **and** solving a linear system.
  - Matrix multiplication only requires “addition” and “multiplication”
  - Solving requires additional operations: “subtraction” and “division”
- A user wishing to perform matrix multiplication on their derived type might be forced to invent suitable operations for subtraction and division.

- Encapsulates relationships among deferred parameters
- Parameterized by deferred arguments, just as with templates
- Declaration of deferred arguments within a REQUIREMENT construct induce

```
REQUIREMENT simple_binop{op, T}
  TYPE, DEFERRED :: T
  INTERFACE
    FUNCTION op(x, y) RESULT(z)
       TYPE(T), INTENT(IN) :: x, y
       TYPE(T) :: z
    END FUNCTION
  END INTERFACE
END REQUIREMENT
```

```
REQUIREMENT binop{op, T, U, V}
  TYPE, DEFERRED :: T, U, V
  INTERFACE
    FUNCTION op(x, y) RESULT(z)
       TYPE(T), INTENT(IN) :: x
       TYPE(U), INTENT(IN) :: y
       TYPE(V) :: z
    END FUNCTION
  END INTERFACE
END REQUIREMENT
```

- The REQUIRES statement applies a named REQUIREMENT to its arguments.

```
TEMPLATE my_templ(T, U, plus, times)
  REQUIRES binop{plus, T, U, U} ! Real+complex -> complex
  REQUIRES binop{times, T, U, U} ! Real*complex -> complex

…
END TEMPLATE


INSTANTIATE my_tmpl{REAL, REAL, operator(+), operator(*)} ! Ok
INSTANTIATE my_tmpl{REAL, LOGICAL, operator(+), operator(*)} ! invalid
```

- The INSTANTIATE statement specifies an instance of a template construct
  - By default, brings all public entities of the template instance into the current scope.
  - Can use ONLY clause for greater control and renaming; just as with module USE statement
- Instantiation args become associated with the deferred args of template
  - Supports keyword association just as with procedure references.

```
INSTANTIATE my_tmpl{T=REAL, n=3, op=OPERATOR(*)}
```

- A *simple template procedure* – concise syntax for declaring (and instantiating) template that has a single procedure

```
TEMPLATE swap{T}(x, y)
          TYPE, DEFERRED :: T
          TYPE(T), INTENT(INOUT) :: x, y
          TYPE(T) :: tmp

          tmp = x
          x = y
          y = x
END TEMPLATE

REAL :: x, y
LOGICAL :: f1, f2

CALL swap{REAL}(x, y)    ! Instantiate for T ←→ REAL
CALL swap{MyType}(q1,q2) ! Instantiate for T ←→ MyType
```

“LFortran is a modern open-source (BSD licensed) interactive Fortran compiler built on top of LLVM. It can execute user’s code interactively to allow exploratory work (much like Python, MATLAB or Julia) as well as compile to binaries with the goal to run user’s code on modern architectures such as multi-core CPUs and GPUs.”

- LFortran has prototyped many of the generic programming features

- Subgroup needs to wait to see where the pain points are

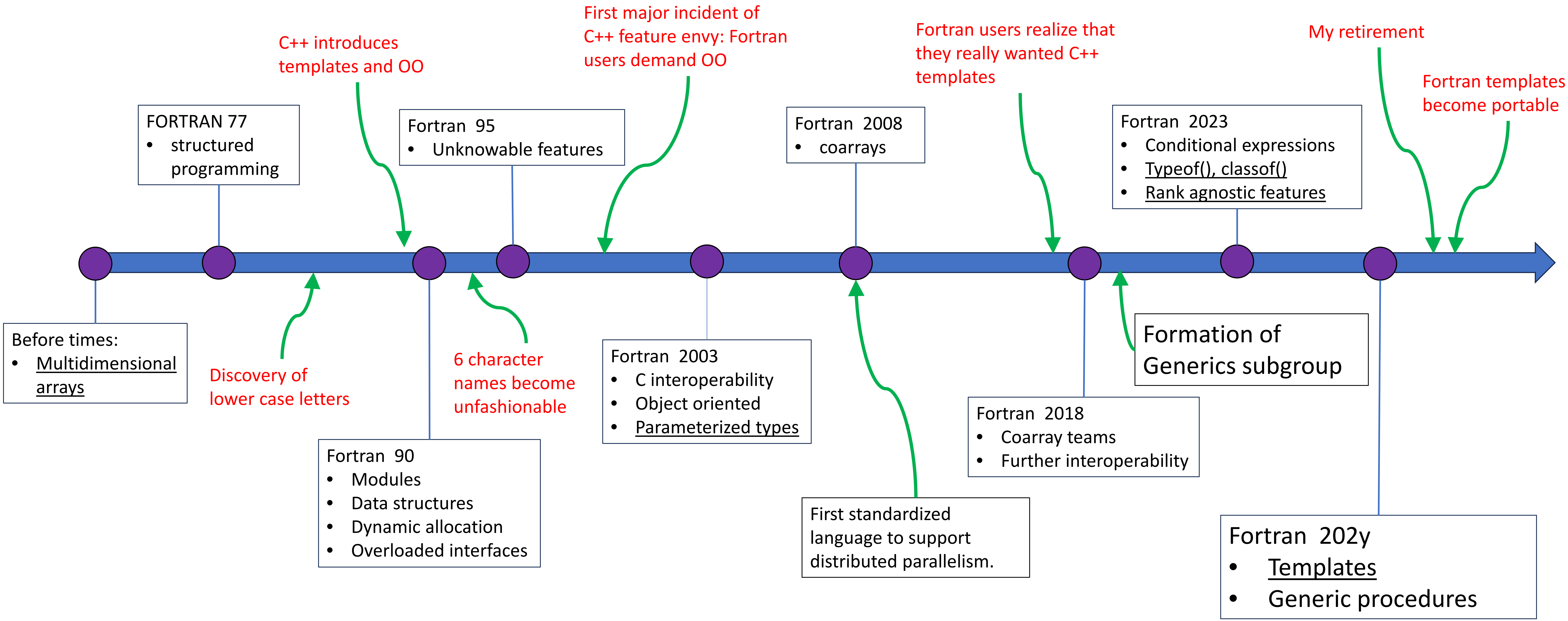
- Better support for type-bound procedures
- Long argument lists
- Other types as constants
- Intrinsic requirements?

```
REQUIRE (M == N + 1)
REQUIRE (ANY(K == REAL_KINDS))
REQUIRE (EXTENDS(T, U))
```

- M. Haveraaen, J. Jaarvi, & D. Rouson,“Reflecting on Generics for Fortran”, https://j3-fortran.org/doc/year/19/19-188.pdf.
- O. Certik, “LFortran - Modern interactive LLVM-based Fortran compiler”, https://lfortran.org.

Supplemental slides

Please do not use for content



Before times:
- Multidimensional arrays

FORTRAN 77
- structured programming

Discovery of lower case letters

C++ introduces templates and OO

Fortran 90
- Modules
- Data structures
- Dynamic allocation
- Overloaded interfaces

6 character names become unfashionable

Fortran 95
- Unknowable features

First major incident of C++ feature envy: Fortran users demand OO

Fortran 2003
- C interoperability
- Object oriented
- Parameterized types

Fortran 2008
- coarrays

First standardized language to support distributed parallelism.

Fortran users realize that they really wanted C++ templates

Fortran 2018
- Coarray teams
- Further interoperability

Formation of Generics subgroup

Fortran 2023
- Conditional expressions
- Typeof(), classof()
- Rank agnostic features

Fortran 202y
- Templates
- Generic procedures

My retirement

Fortran templates become portable

- **Insight** A generic mechanism needs to allow types and operations as arguments.
- **Insight** A generic mechanism needs to treat intrinsics in the same way as user-defined types and operations.
- **Insight** Generic parameters, code and arguments should be fully typechecked.
- **Insight** Generics features should be straightforward (careful with overloading and specialisa- tion) enough to allow for straightforward modular typechecking of generic code.
- **Insight** A generic mechanism must allow nested parameter lists to match its code structuring mechanisms.
- **Insight** A generic mechanism must support writing long parameter lists.
- **Insight** A generic mechanism must support renaming of generic types and operations.
- **Insight** A generic mechanism must not depend on statically declared inheritance hierarchies.
- **Insight** This shows that not only types but also operations (functions and subroutines), both user-defined and intrinsic, are needed as generic arguments2.
- **Insight** typechecking of generic code ensures type safe expressions, guaranteeing well-formedness properties of the code.
- **Insight** Long generic argument lists enable expressing fine-grained type safety without having to extend the type system.
- **Insight** Both types and operations with prototypes are needed as generic arguments.
- **Insight** Long generic argument lists with types and operation prototypes with full typechecking of generic code enables proof by testing.
- **Insight** Long generic argument lists with types and operation prototypes with full typechecking of generic code enables iterating generic implementations.
- **Insight** Generic programming may benefit from an axiom based support system for keeping track of properties of generic constructions.
- **Insight** A renaming mechanism along with types and operations as generic arguments is sufficient to enable a future extension of generics with axioms.

- **Insight** A generic mechanism needs to allow types and operations as arguments.
- **Insight** A generic mechanism needs to treat intrinsics in the same way as user-defined types and operations.
- **Insight** Generic parameters, code and arguments should be fully typechecked.
- **Insight** Generics features should be straightforward (careful with overloading and specialisa- tion) enough to allow for straightforward modular typechecking of generic code.
- **Insight** A generic mechanism must allow nested parameter lists to match its code structuring mechanisms.
- **Insight** A generic mechanism must support writing long parameter lists.
- **Insight** A generic mechanism must support renaming of generic types and operations.
- **Insight** A generic mechanism must not depend on statically declared inheritance hierarchies.
- **Insight** This shows that not only types but also operations (functions and subroutines), both user-defined and intrinsic, are needed as generic arguments2.
- **Insight** typechecking of generic code ensures type safe expressions, guaranteeing well-formedness properties of the code.
- **Insight** Long generic argument lists enable expressing fine-grained type safety without having to extend the type system.
- **Insight** Both types and operations with prototypes are needed as generic arguments.
- **Insight** Long generic argument lists with types and operation prototypes with full typechecking of generic code enables proof by testing.
- **Insight** Long generic argument lists with types and operation prototypes with full typechecking of generic code enables iterating generic implementations.
- **Insight** Generic programming may benefit from an axiom based support system for keeping track of properties of generic constructions.
- **Insight** A renaming mechanism along with types and operations as generic arguments is sufficient to enable a future extension of generics with axioms.

```
TEMPLATE my_template(T, U, C, F)
  TYPE, DEFERRED :: T, U
  INTEGER, CONSTANT :: C
  INTERFACE
    FUNCTION F(x) RESULT(Y)
      TYPE(T) :: x
      TYPE(U) :: y
    END FUNCTION
  END INTERFACE
CONTAINS
  SUBROUTINE do_something(x, y, z)
   TYPE(T) :: x, y
   TYPE(U) :: z
   . . .
  END SUBROUTINE
END TEMPLATE
```

```
REQUIREMENT BINARY_OP(T, op)
  TYPE, DEFERRED :: T
  INTERFACE
    FUNCTION op(x) RESULT(Y)
      TYPE(T), INTENT(IN) :: x
      TYPE(T) :: y
    END FUNCTION
  END INTERFACE
END REQUIREMENT

TEMPLATE tmpl(T, U, f1, f2)
  REQUIRES BINARY_OP(T, f1)
  REQUIRES BINARY_OP(T, f2)
  . . .
END TEMPLATE
```