Reflecting on Generics for Fortran

Magne Haveraaen* Jaakko Järvi* Damian Rouson†

July 23, 2019

Abstract

This paper focuses on the benefits of a well-designed generics feature, and points out important design decisions for achieving those benefits.

1. A generics facility needs to be type safe, both for the developer and the user of generic code.
2. Built in (intrinsics) and user defined (derived) types and operations (functions/subroutines) need to be treated in the same way as generic arguments.
3. A generics facility needs to allow generic parameter lists to be extended/nested.

The note does not discuss technicalities of a specific generics design. It is motivated by the upcoming Fortran discussion in Tokyo (JP) in August 2019.

1 Introduction

For the upcoming IEC/ISO WG5 & ANSII J3 Fortran committee meeting in Tokyo August 5-9 there are several proposals for adding a generics feature. Fortran currently lacks a generic mechanism to allow type and operation parameters to modules, classes and operations (functions and subroutines). This prevents Fortran from providing, e.g., reusable collection classes which are now a standard feature of most other mainstream programming languages.

The design space of generics is large, evidenced by how different the generics features of different languages are (see some examples in Appendix A). This paper emphasises a few key demands for such a feature, based on observations of problems and successes of generics in various languages. These demands are crucial if one wishes to unleash the full potential of generic code reuse, going beyond the standard examples of generics (collection classes, etc.). A generics feature should also be simple to implement. None of the requirements below will make implementing generics more difficult. Ignoring them will on the other hand limit the adoption and usefulness of a generics feature.

- Formal generic parameters should be types, operations with full prototype declarations or typed values, and generic code must be fully typechecked against these constraints (see Section 2.3).

  This helps developers of generic code catch simple errors early, and avoids postponing their discovery to instantiation time – a severe problem with the C++ template mechanism. It also enables techniques for fine-grained typechecking (e.g., similar to equipping numeric types with units and checking unit correctness, see Section 3.2), more reusable algorithms (e.g., an algorithm that operates on an array through an abstract concept of an index type cannot accidentally rely on the indices happening to be of a particular integral type), and generic parametricity enable proofs of correctness by testing of generic code (see Section 3.3).

  Typed parameter constraints and typechecking of generic code also allows for different compilation models for generics, whether to expand each instance of a generic definition (C++) or generate just one shared code for all instances (Java, Ada).

*Bergen Language Design Laboratory (BLDL), Department of Informatics, University of Bergen, Norway
†Sourcery institute, Oakland, California, USA

- Generic arguments should allow both intrinsic and user defined (derived) types and operations, and the instantiation should be fully checked against the prototype declarations.

  This ensures flexible reuse (see Section 2.2) and error free instantiation of generics (see Section 2.3). The latter avoids instantiation errors as those experienced with the C++ template mechanism, which C++ now is trying to rein in by the proposed "concept lite" mechanism.

- Nested generic definitions should be allowed (e.g., a generic module inside a generic module).

  This allows for flexible structuring of generic definitions (see Section 2.4).

- In the using context, when instantiating generic code, it should be possible to rename the defined types and operations.

  Renaming of defined entities when instantiating generic definitions is important for flexible reuse (see Section 2.5).

- Reusable renaming lists should be supported.

  Generic parameter lists do get long, and very often the similar instantiations will occur for generic code. Making these renaming lists reusable will simplify the use of advanced generic code (see Section 2.5). Similar reuse of renaming lists for instantiated generic code further enhances the usability of complex generic code (see Section 3.4).

A well-defined generics feature that meets these demands enables *generic programming* for plain programmers and not just for experts. Generic programming has been explored by Alex Stepanov [8], the developer of the C++ Standard Template Library (STL). The essence of this programming paradigm is that the capabilities (operations and types) that algorithms need to assume from their parameters are formalised as *concepts*, and generic functions are implemented in terms of these concepts. The approach has shown to lead to highly reusable libraries. When the STL was introduced, Bjarne Stroustrup considered generic programming a disruptive programming technique—it is now considered the *modern approach* to C++ programming [9]. The requirements above avoid the pitfalls that often get associated with the C++ template mechanism (the difficulty to get templates type correct, and the possibly large error messages appearing due to template instantiation or template implementation errors), while enabling more advanced uses of generics not easily exploitable with the C++ template mechanism.

2 Design Considerations for a Generics Feature

Many languages that support generics lack one or more of the requirements discussed above, making it difficult to develop correct generic code or use it in a correct way, with the consequence that the language cannot draw the full potential of generics.

- Java: does not support the use of language defined types (intrinsics) as generic arguments, only class types are allowed. Java's *erasure* semantics of compiling generics creates further restrictions and gotchas on the use of generic types (`instanceof` not making a distinction between different instantiations of the same generic class, effects of *bridge methods*[1] etc.). Another problem is that classes must declare which interfaces they implement at the point of definition of the class, which precludes generic reuse without the use of wrapper classes.

- C++: does not typecheck template bodies. It only typechecks the actual instantiation, which often leads to error messages that are very difficult to decipher. The newly added *concept lite* construct is designed to alleviate the problems of complex error messages for incorrect use of templates, but it does not address typechecking of template bodies.

[1] `https://docs.oracle.com/javase/tutorial/java/generics/bridgeMethods.html`

- ML: supports long generic argument lists through signature declarations and use. The drawback is that often distinct signature arguments are intended to share type and operation declarations. This must then be handled by explicit "sharing" annotations.

In the following, we discuss individual aspects of the design space of generics in more detail. The focus is on the benefits of particular design choices, not technicalities of their possible manifestation in Fortran.

2.1 Semantics via substitution

In general a generic mechanism is semantically very simple. Generic code instantiation can be understood simply as the substitution of the generic arguments into the generic code body. In this paper we argue for a generics feature with such semantics: generic code can use generic type names and function names, and require that all instantiations substitute types and (correctly typed) functions for those names. Many of the problems of generics features stem from languages deviating from these simple semantics.

2.2 Instantiate with intrinsics and user-defined types and operations

Traditionally a programming language is for pragmatical reasons often split into the types and operations that belong to the language (intrinsics) and those that can be defined by a "user" library. The notion of a "user library" encompasses the language's standard library, third party libraries (often vendor specific), and end users of the language (application programmers). The standard library may extend a language with features that are not user implementable (requiring special support from the compiler) but nevertheless behave similarly to user-definable types and operations. Likewise, third party libraries may have special code for specific hardware and use architecture specific code, still behaving as user-definable.

The challenge for a generic facility is to not differentiate between built-in features and user code. Any disparity here will cause significant problems for code reuse down the line. A common example illustrating such problems is given by Java, where classes and operations may be generic on (user-definable) types and operations, while the same operations on built in types need to be overloaded on a case by case basis (same source code, only the typing differs). This prevents, e.g., a generic stack implementation to be reused for integer and float element types. Java solved this by introducing special classes corresponding to each built in type, and providing automatic conversion between the builtin type and the corresponding class. The solution is only partial: the class version lacks the wrapped type's operations, e.g., comparison operations. Thus the number classes cannot be used as elements of binary search trees. Users can solve this by creating their own wrapper classes for each builtin type and adding the needed operations, but then the automatic conversion between builtin type and the corresponding class is lost. Generics not handling intrinsic and user-defined types in the same way causes layers and layers of problems.

Generic data structures and algorithms do not only depend on the type of data they work on, but also on specific operations on that type. The operations can be built-in or user-defined functions. Consider a sorting algorithm parameterised by the comparison function between two elements: instantiations should be allowed to bind it to, say, the built-in less-than comparison operator for integers, or to any user defined comparison operator.

An example is the FORTRAN 2008 `FINDLOC` intrinsic function that locates an element within an array given certain criteria. This utility function is only available for arrays of intrinsic types. From a user perspective, this is an artificial restriction. A generic definition of such a function relies on being able to compare the element type with an equality operator and it would be usable with any type, whether intrinsic or derived, as long as there was a way to instantiate the procedure with a comparison operation for that particular type. Another benefit of generics, is that this function then easily can be implemented by the user, if it is not available for a specific compiler. Further, a third party library could provide such a function across all compilers, reducing the need for compiler vendors to implement it. Such a third party library could also use special

optimisation tricks, e.g., vectorisation, given certain combinations of argument types for selected target architectures.

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.

2.3 Typechecking of generics

Modular typechecking of the body of generic code necessitates that generic definitions include declarations of parameter prototypes for generic types and operations. Modular typechecking means that

- the body of the generic code is checked against the prototype declarations: generic code can only use types and operations that are declared; and
- types and operations that clients instantiate generic code with are checked against the prototype declarations: all types and operations declared in the prototypes must be defined.

Modular typechecking is very important for both developers and clients of generic libraries:

- library developers can be sure that the type system to reject their generic code, if they accidentally rely on features that are not provided by the declared generic prototypes;
- developers of client code can be sure that their code is rejected if they try to use generic code with parameters that do not satisfy the declared generic prototypes—and if they do, there will be no type errors arising from the generic code (pointing to internals of a generic library).

C++ traditionally has had neither forms of typechecking; generic prototypes have only been defined by documentation, not enforced by code. As a result, modern complex C++ template libraries are notorious for their extremely long and hard to decipher error messages that result from simple mistakes in template instantiations.

The (likely) forthcoming "concept lite" feature will add typechecking of client code, template instantiations, against the generic prototypes—but not typechecking of bodies of generic definitions. If the generic prototype declarations and the generic body do not match, clients may still get obscure error messages that point to library internals. The design choice of only typechecking the clients of generic code was made because there was no consensus on details of typechecking generic code in the presence of overloading and template specialisation.

Java and Haskell, for example, support modular typechecking. Java's type erasure semantics imposes (sometimes seemingly quite arbitrary) restrictions on generic code to achieve modular typechecking. In Haskell, the desire to be able to infer types is what impose restrictions on generic code. Several rather complex features have been added to the language (functional dependencies, associated types, type families, etc.) to lift those restrictions, but it is probably fair to say that their use requires quite in-depth knowledge of Haskell.

Insight Generic parameters, code and arguments should be fully typechecked.

Insight Generics features should be straightforward (careful with overloading and specialisation) enough to allow for straightforward modular typechecking of generic code.

2.4 Nested generic argument lists

When a language allows nested constructs, e.g., inner classes or operations inside generic modules, the inner construct will in some cases need to extend the generic parameter list from the enclosing construct.

Consider a generic class implementing sparse matrices with a certain layout for any number type. Such a class may also provide reduction operators, e.g., returning the maximum (reduction over `max`) or the minimum element (reduction over `min`), or the sum of all elements (reduction

over +). A reduction operator can be implemented by a function taking the binary operation as a generic argument, extending the generic parameter list of the containing sparse array class.

Insight A generic mechanism must allow nested parameter lists to match its code structuring mechanisms.

2.5 Long generic argument lists and renaming

When embracing the generic programming approach to library development, the number of parameters in generic definitions can become large. If positional parameter lists are the only way of expressing parameterisation, things get out of hands quickly. For instance, a library for polynomials can be defined based on a ring interface (one type and five operations). To list all these parameters explicitly (at every use of a generic definition) in a generic library would be unwieldy. Instead, a typical C++ template library would fix the names of the five operations $(+, -, 0, *, 1)$ and attach them to the one type argument. This severely limits the use of such a template polynomial library. For instance, matrices have two multiplication operations (matrix multiplication and pointwise multiplication) that provide matrices with a ring structure, but only one of these (the one with the name $*$) can be used as an argument for a template polynomial library with a single type parameter and no operation parameters. We would need to use tricks with encapsulation and wrappers in order to use the other ring structure as an argument to the library.

The name-based renaming used in Fortran module interface declarations is a better syntactic mechanism. Possibly extending this with reusable renaming lists is needed since with long renaming lists many (syntactically) similar instantiations will occur in practice. Reusing a renaming will facilitate simpler reuse of long parameter lists in similar circumstances.

It turns out that in practice the same type or operation names are defined in many modules. One way of handling this is renaming of defined types and operations to more convenient names in the using context. C++ allows this for types with the `typedef` and the recent and more flexible *alias* mechanism, but for operations the only mechanism is wrappers, with the expectation that the compiler will inline them away. A renaming mechanism may be extended to cover also this need.

Insight A generic mechanism must support writing long parameter lists.

Insight A generic mechanism must support renaming of generic types and operations.

2.6 Generics and inheritance hierarchies

Java has tied their generics mechanism, the interface, to the inheritance hierarchy. Any class may inherit from a list of interfaces. An implementation may declare an interface as a generic parameter. Then any class that explicitly implements that interface can be used as a generic parameter. The problem is that a user cannot retrofit a new interface to existing classes from the standard library or third party libraries. For instance, a reduction operator on arrays may take a user-defined `Monoid` interface as a generic parameter. Since no existing class will implement this interface, the programmer has to wrap existing classes into a new hierarchy implementing `Monoid`. Doing this over and over again for all relevant existing classes and every new user-defined generic argument list is not sustainable and limits the practical use of generics.

Insight A generic mechanism must not depend on statically declared inheritance hierarchies.

2.7 Implementing generics and fear of code bloat

A point against Fortran generics has been the fear of code bloat: large executables as a result of generating essentially the same code for different instances of generic functions. This has been observed in languages like C++, where different template instantiations often appear explicitly in the compiled code. This has been countered, e.g., by Ada, where the language design avoids code bloat by the compiler. Even in C++ code bloat is seldom a significant issue: compilers do not generate code for those methods of a generic class that are not called, and modern compilers/linkers recognise when two instances produce identical code, and share it.

A more important problem, however, is the *source code bloat* that software developers experience when having to implement the same source code for different combinations of types and operations. Such duplication leads to huge maintenance problems, far more significant than a possible code bloat effect for compiled code.

We emphasise that the simple substitution semantics of generics advocated in this paper is agnostic to how generics is implemented. Compilers can perform substitution at the AST level as in C++ (or even at source text level), or generate just one piece of executable code for each generic definition and at their call sites construct dictionaries of function pointers to be passed in as additional arguments (as in Haskell, usually). As speed/size optimisations, a compiler might, for example, choose to perform concrete substitution for generic definitions that are instantiated only a small number of times but use dictionary passing for definitions that are instantiated with many different types.

2.8 A remark about metaprogramming

Metaprogramming and generics are two orthogonal features of a programming language.

Generics has to do with reuse of code, and is a conceptually simple mechanism in which code can be understood semantically by substituting the instantiating arguments in place of the formal generic arguments.

Metaprogramming deals with generating source code. Many languages, from Lisp onwards, now including Java, has runtime features for accessing some aspects of the source code and letting the user program generate new code. Static reflection is when the compiler interprets parts of the source code, at compile time, on some term representation of some part of the remaining source code, in order to modify the code that is generated.

C++ has become famous for "template metaprogramming", where by coincidence the type system of C++ turned out to be powerful enough to allow compile-time reflection. This is a very complicated to use metaprogramming facility. For one, the template type level compile-time interpreted language is very cumbersome to use. Second, such metaprogramming only applies to single expressions in the source code.

Interestingly, with a sufficiently powerful generic mechanism, it is possible to do staged metaprogramming in any language without further support from the programming language. Staged metaprogramming means that the compiler (1) emits code that, (2) when run, builds a term tree representing some aspect of the code. The parse tree is then (3) manipulated by a normal program, and the resulting modified parse tree is then printed as source code. This created source is then (4) compiled by the compiler to yield the intended executable. Though staged metaprogramming requires multiple steps, they can all be captured in a makefile, in practice giving "compile-time" metaprogramming.

3 Benefits of Generics

Here we go deeper into the benefits a well designed generics facility may give the software developer. In addition to the standard, well-known, collection class examples, we also show some cutting edge uses of generics. The benefits of generics have a profound impact on code quality and code design.

Here we look at the standard generic use cases (collection classes), but also explore further benefits given type safety of generic code with the ability to handle long generic parameter lists and reusable renaming lists.

3.1 Reusable Collection Classes

The standard example of benefits from generics is the ability to provide a reusable library of collection classes. The simple cases require just a type as parameter, and implicitly assume assignment is available for all types.

- Lists, arrays, matrices,

- Trees,
- Graphs.

Insight In order to instantiate these collection classes with any relevant type, the generic mechanism cannot differentiate between user-defined and intrinsic types.

Assuming the element type comes with equality comparison, it is possible to search for data placed inside such structures, and get a pointer/path to the data item if it is there. More advanced collection class implementations assume the availability of a comparison operation (should be a partial or total order).

- Sorted lists
- Binary search trees, tries, etc
- Sorting algorithms, binary search in arrays, etc.

A more efficient store/search collection data structure requires a hash function in addition to equality comparison.

- Hash tables.

Insight This shows that not only types but also operations (functions and subroutines), both user-defined and intrinsic, are needed as generic arguments[2].

Once a basic generic mechanism is in place, it becomes possible to introduce collection classes as part of a standard library, or let third parties define (and share) their own take on those abstractions.

3.2 Exploiting type safety in generic code

If the syntax for generics supports very long argument lists, typechecking of operators in the body can be used to increase code safety. We saw this trick first presented in the financial domain, so the motivating example is taken from there.

The idea is to introduce a separate type for each currency, the interest rate in each currency, and the conversion rate between two currencies. By keeping them as separate types, the typechecker prevents us from confusing the different types of data and mixing up the numbers, such as mistaking the conversion rate for the interest rate deep down in some computation. The example below is written in the programming language Magnolia, developed at BLDL.

```
1  /** Declares currency X with interest rate XIR. */
2  signature Currency = {
3    type X;
4    type XIR;
5    function accrue ( amount:X, rate:XIR ) : X;
6  };
7  /** Declares currency conversion from X to Y. */
8  signature Conversion = {
9    type X;
10   type Y;
11   /** Conversion rate. */
12   type X_to_Y;
13   function convert ( amount:X, rate:X_to_Y ) : Y;
14 };
```

[2]For these simple examples functional programmers often argue that higher-order function parameters (e.g., pass the comparison operation as a run-time argument in the constructor) are sufficient. However, higher order functions have several drawbacks. They prohibit the compiler from generating optimal code based on the knowledge of a statically bound generic argument since higher-order arguments are runtime oriented. Using higher-order arguments runs in to the previously discussed problems regarding long generic parameter lists.

```
/** Computes profit for amount in currency X, either
 * by saving funds as X or by converting and saving them as Y.
 */
implementation profit = {
  require Currency;
  require Currency [ X => Y, XIR => YIR ];
  require Conversion;
  require Conversion [ X => Y, Y => X, X_to_Y => Y_to_X ];

  function profitX ( amountX:X, interestX:XIR ) : X
  = accrue(amountX,interestX);
  function profitY
    ( amountX:X, interestY:YIR, rateXY:X_to_Y, rateYX:Y_to_X ) : X
  = convert(accrue(convert(amountX,rateXY),interestY),rateYX);
};
/** Renaming the instantiation of X to NOK, and Y to USD. */
renaming currency_NOK_USD =
    [ X => NOK, XIR => NOKIR, profitX => profitNOK ]
    [ Y => USD, YIR => USDIR, profitY => profitUSD ]
    [ X_to_Y => NOK_to_USD, Y_to_X => USD_to_NOK ]
      ;
/** An implementation using the new notation. */
implementation profit_NOK_USD = profit[ currency_NOK_USD ];
```

typechecking the above, we know all the formulas computing the profit are working with the relevant data. Thus there is no mix up between amount and rates, between rates in one currency or the other, or between conversion one way or the other way.

The module notation above uses the following ideas.

- Declarations of types and function profiles can be collected in named **signature**s (as in ML). These are included were needed (**use** or **require**). In both cases the union of all included declarations are taken, and identical type names and function declarations are coalesced. This avoids the need for ML style "sharing" declarations. Such coalescing is common in algebraic specification languages [7].
- In an implementation ("module" in Fortran speak), the **require**d declarations are the generic parameters.
- Declarations (and implementations) can be adopted to new contexts by renaming the required entities. In the above example, in implementation `profit`, first the currency X and rate XIR are renamed to Y and YIR in the second inclusion of `Currency`, and then X and Y trade places in the second inclusion of `Conversion`. Then in the implementation `profit_NOK_USD` X and Y are further renamed to NOK[3] and USD[4], respectively. Also some of the functions are renamed to let the name aid as a mnemonic for which currency is being handled.

The implementation `profit_NOK_USD` corresponds to the following declarations and code.

```
/** Expanded code corresponding to profit_NOK_USD above. */
implementation profit_NOK_USD_expanded = {
// Defined
  function profitNOK ( amountX:NOK, interestX:NOKIR ) : NOK
  = accrue(amountX,interestX);
  function profitUSD
```

[3]The Norwegian currency
[4]The currency used in USA

```
    ( amountX:NOK, interestY:USDIR, rateXY:NOK_to_USD, rateYX:USD_to_NOK ) : NOK
  = convert(accrue(convert(amountX,rateXY),interestY),rateYX);

// Required
  type NOK;
  type NOKIR;
  type NOK_to_USD;
  type USD;
  type USDIR;
  type USD_to_NOK;
  predicate _<=_ ( a:NOK, b:NOK );
  predicate _<=_ ( a:USD, b:USD );
  function accrue ( amount:NOK, rate:NOKIR ) : NOK;
  function accrue ( amount:USD, rate:USDIR ) : USD;
  function convert ( amount:NOK, rate:NOK_to_USD ) : USD;
  function convert ( amount:USD, rate:USD_to_NOK ) : NOK;
};
```

Finally we will instantiate all of the types to a high precision `REAL(10)` type for doing the actual computations. In the instantiation context we no longer have the benefit of the separated types to avoid data confusion.

```
/** Instantiating a program using high precision floats and multiplications. */
program profitCxx = {
  use float80bitCxx;
  use profit_NOK_USD
    [ NOK => Float, NOKIR => Float ]
    [ USD => Float, USDIR => Float,
      NOK_to_USD => Float, USD_to_NOK => Float ]
    [ accrue => _*_, convert => _*_ ];
};
```

The code that is in this unsafe space is significantly shorter than the safe code, and it does not deal with getting the profit formulas correct (those are in the safe part of the code). Note how changing the functions and operations on instantiation keeps type safety, but effectively removes all overhead compared to the alternative, which would be to use wrapper types: the instantiation will compute directly on floating point using the intrinsic multiplication and addition operations.

```
/** Expanded code corresponding to profitCxx above. */
program profitCxx_expanded = {
  use float80bitCxx;
  function profitNOK ( amountX:Float, interestX:Float ) : Float
  = amountX * interestX;
  function profitUSD ( amountX:Float, interestY:Float,
                       rateXY:Float, rateYX:Float ) : Float;
  function profitUSD
    ( amountX:Float, interestY:Float, rateXY:Float, rateYX:Float ) : Float
  = amountX * rateXY * interestY * rateYX;
};
```

When dealing with a domain, this exploitation of generics has a similar effect to that of introducing units (here, user defined monetary units like NOK, USD, NOK_IR, USD_IR, NOK_to_USD_rate, USD_to_NOK_rate, etc) on type safety. It does so within a uniform mechanism (generics), rather than needing to add another mechanism (a unit system with user-definable units) to the language.

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.

3.3 Proof by testing

The type safety guarantees discussed above can be exploited for *proving* correctness of code.

As an example consider a library created for cache optimal traversal of data. Such a library will have many special cases, where the size of the different cache levels interact with the size of the data set, giving many specific traversal patterns and unfolding parts of loops for initialising pipelines. Thus even if the library has been carefully written and well tested, the question always remains: will it work correctly for "my problem"?

So being cautious software developers we need to do one of

- trust that the library writers did verify their code so it is guaranteed to work for all cases,
- try to verify the library code ourselves (if the source is available), or
- do extensive testing of the library ourselves to gain confidence.

Here we show that if the library is sufficiently generic, it is possible to prove the library correct for "my problem" with a few carefully chosen test cases.

Let us consider a function `map` that takes a binary function `bin` (generic argument to `map`) and two arrays. The specification says that the statement `a = map(a1,a2);` should have the effect that `a(i) == bin(a1(i),a2(i))` for all valid indices `i`. Assume that for some special index value `k` inside the library the setting of `a(k)` is wrong. Thus `a(k)` can be assigned a value that is

1. some arbitrary constant,
2. some irrelevant value dependent on `k`, for instance `k` itself,
3. some wrongly indexed value, for instance `a1(i)` for some valid index `i`,
4. some mix up of arguments to `bin`, for instance `bin(a2(i),a2(i))`, or
5. some mix up of the indexing of the arguments to `bin` as called in the library, for instance `bin(a1(i),a2(j))` for `i` or `j` (or both) different from `k`.

The error may happen in just a few cases for special combinations of cache and data sizes. Thus the error is very difficult to discover by testing, and it may easily be overlooked when carefully reading the source code.

Interestingly, such problems are more easily detected the more generic the library is, given that the generic library is strongly typed. Let us consider three cases of genericity.

- The data element types are fixed to real numbers (float), and only `bin` (on reals) is a generic argument (or a function pointer as in classical C).

  This is the classical case, where all error situations sketched above are possible. Even with careful code review and extensive testing we can never be sure we did not miss a case. The effective countermeasure against such errors is a full proof of correctness.

- The data element types are of a generic type `T`, and `bin` is generic in `T`.

  This typing regime eliminates the two first error variants as a possibility: the library does not know the type `T`, thus assigning a constant value or some other expression not involving the arrays and `bin` is not possible. Eliminating the three latter kinds of errors still requires careful code reading, testing, and possibly formal verification.

- The data element types are of generic types `T`, `T1` `T2`, and `bin` is generic in taking a `T1` (first argument) a `T2` (second argument), returning a value of type `T`.

  Strong typing now eliminates all but the last case of errors from the body of the `map` function: typing forces an element value from array `a1` to be used as left argument to `bin`, an element value from array `a2` to be used as right argument to `bin`, and the output from calling `bin` is the only value type that can be assigned to `a`.

Note how we increase the generic flexibility as we move downward in this list: the library becomes more reusable as more use combinations are becoming available, seemingly also drastically increasing the test domain. On the other side, the more generic the code becomes, the error possibilities are cornered by the type system. This effect is called *parametricity*, and it gives fundamental insights on generic code [10, 4]. The more parametric, i.e., the more finegrained the generic arguments are, the more knowledge we gain, given that the generic code is fully typechecked.

For the example generic `map` function this gives us *proof by testing* for "my problem". The algorithm for `map` may be a wilderness of (type correct) special cases, but once we know the size of "my" data set and the cache sizes of "my" machine, the map algorithm is fixed and completely independent of the choice of the generic arguments. Thus however convoluted the library implementation is, the only possible error is that `a(k) = bin(a1(i),a2(j))` for inappropriate `i` and `j`. One test case that can discover any such error is instantiating all types to REAL (float) and `bin` to multiplication. Then filling the argument arrays with distinct prime numbers will detect any misindexing error, since the expected value of the product of two primes will be unique for every element `a(k)`.

Thus even if the library contains erroneous special cases for other combinations of data sizes and cache sizes, we have *proven* its correctness for "my problem" by this test case. The test for correctness will need to be repeated whenever the library, the data size or the cache size (different machine) is changed. However, the cost of such a test for correctness is negligible compared to the actual computations that are to be carried out. Installing such a test regime has a minimal cost for ensuring that the library is correct for "my problem". More details can be found in the paper [6].

Insight Long generic argument lists with types and operation prototypes with full typechecking of generic code enables *proof by testing*.

3.4 Iterating generic implementations

There are many cases in mathematics where we have simple constructions that we can apply repeatedly, and each application builds a more complex mathematical structure. Some well known examples are:

- The Cayley–Dickson construction is a generic code that takes any structure with $+, -, 0, *, /, 1$ and conjugate to a more advanced structure with the same operations. This allows us to construct the complex numbers from the real numbers, the quaternions from the complex numbers, the octonions from the quaternions, and so forth. Without / in the API, we can use this same construction to build the Gaussian numbers from the integers.

  There are also theorems about which properties are conserved by the construction. The group aspect of $+, -, 0$ is always preserved, so each of the constructions will give a vector space of twice the dimensions of the source vector space (1 for the reals, 2 for complex, 4 for quaternions,...). Distributivity is always preserved, and so is the neutral property of 1 with respect to $*$. However, the associativity of $*$ quickly deteriorates. Reals (the conjugate is trivial) and complex have associative $*$ (the conjugate is non-trivial), quaternions have alternation ($x(xy) = (xx)y$ and $x(yy) = (xy)y$), while further constructions only preserve power associativity ($x^n * x^m = x^m * x^n$ where $x^0 = 1$ and $x^{n+1} = x^n * x$ for $n \geq 0$). Likewise commutativity of $*$ disappears for quaternions and beyond.

- There are many constructions taking us from the unit ring signature $+, -, 0, *, 1$ to the same signature: constructing direct products (generalising direct sums), constructing fractions

(which also produces /), constructing modules over a ring (a generalisation of vector spaces over a field), constructing linear algebra (matrix algebra), constructing multi-linear algebra (tensor algebra), constructing ring fields (both continuous and discrete), constructing polynomials, etc.

These constructions can then be combined arbitrarily, some combinations maintaining the properties of the underlying algebra, other selectively losing or gaining properties. Sometimes a double construction is isomorphic to a single construction, e.g., fractions of fractions of a ring is isomorphic to a single fraction construction on the ring. Other combinations may commute, e.g., multi-linear algebra of a ring field from a ring is isomorphic to a ring field of multi-linear algebra of the ring. This allows us to consider alternate ways of building abstractions, e.g., abstract solvers using coordinate-free numerics for PDEs.

Going further we may equip the linear algebra construction with a conjugate operation based on the underlying ring. Then we can apply the Cayley–Dickson construction on the linear algebra construction on rings with conjugate, and show that in some cases it is isomorphic to the linear algebra construction of a Cayley–Dickson construction on the ring. Doing linear algebra on complex numbers can then be implemented as doing complex arithmetic on pairs of matrices. This was utilised in early Fortran libraries, before complex was a well supported Fortran data type.

- Moving away from rings we can look at lattice constructions. A lattice has API with signature $\wedge, \vee, \top, \bot$. There are standard constructions on lattices, which given two structures each with a lattice API, yields another lattice structure: the direct product, the lexicographic product, the discrete product, etc.

- The BLDL research group has been working on data dependency algebras (DDAs, a computation oriented API related to directed multigraphs) and identified many constructions yielding DDAs from DDAs [5]. This allows us to build advanced computing structures in a few steps from well understood generic components.

- We can also combine constructions that take us from one domain to another. For instance, from a boolean lattice (lattice with extra operations and structure) we can (generically) construct a boolean ring. A boolean ring is a ring, thus we can use linear algebra to solve problems originally formulated as boolean lattice problems.

Using such constructions allows us to quickly build advanced data structures with relevant operations from simple types. This is an efficient way for developing software. It also allows good control over which properties are maintained by the construction, thus helping us better understand the software we are building. Equally important is that it allows us to verify and test each individual generic code against these expected properties, and then be assured that any instantiation, even the combined ones, behave as expected.

The benefits of exploiting such constructions are large. Without a suitably designed generics facility we are not looking for these constructions, and will therefore not know of their existence. What we need is the ability to use non-trivial lists of types and operations as generic parameters, and having type-safe generic codes and type-safe instantiation of generics.

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.

4 Enhancement of Generics with Axioms

Generic code can be interpreted as a mapping from an API to an API. In the simple, standard cases, this is taking an element type and providing a list of elements, or taking an element type with a comparison predicate and providing a binary search tree. In the latter case we see that we

need properties on the argument predicate, e.g., that it is a total order, for the binary search tree to function as intended.

Properties of generic code can be written as axioms. This is compatible with generic code as API definitions based on API arguments [3]. With some consideration axioms can be added as an extension of a generic facility. Axioms can be expressed in any language with an `assert` facility or an exception mechanism. The essential feature for axiom reuse is a flexible generic instantiation mechanism, since properties like partial order, monoid or ring can be satisfied by many distinct sets of operations for each type.

Insight A renaming mechanism along with types and operations as generic arguments is sufficient to enable a future extension of generics with axioms.

Besides being abstract declarations, axioms can be used for testing and optimisation of code [2, 1]. Using axioms for optimisation purposes is currently heavily discussed in the C++ standardisation committee, and may appear in the 2023 timeframe.

5 Conclusion

Here we have presented necessary requirements on a generics mechanism and some benefits that follow from these.

- It needs to be type safe, both for the writer and the user of generic code.

  Failing this means it will be very difficult for developers to get generic code right, and users of generic code may be left with incomprehensible error messages if something goes wrong.

- Intrinsics and user defined types and functions/subroutines must be treated in the same way.

  Failing this limits reuse of generic code. It will drive developers to maintain multiple versions of the same code, exactly what generics is trying to avoid.

- It must allow generic parameter lists to be extended/nested.

  Failing this will limit generic reuse and modular development of code.

- A modular renaming scheme for instantiating and using generic code.

  Failing this means using generics will be more clumsy.

The benefits that follow from a well designed generics feature are significant.

- Reusable collection classes in line with most other major programming languages.
- Improved code safety by exploiting fine-grained type safety in generic code without hurting code efficiency.
- Proof by testing in the user context by exploiting fine-grained type safety.
- Quickly constructing advanced code by iterating generic code.

Many of these benefits are hard to obtain in other generic programming languages due to them not fulfilling the requirements above.

A Generics in other languages

A.1 Haskell Type Classes

Haskell's mechanism for generics is *type classes*. A type class is parametrized over one or more types, and it defines a collection of function signatures. Type classes can be used as constraints of type parameters of functions. An *instance declaration* posits that certain types satisfy the requirements of a type class.

The following `Currency` class is parametrized by two types `x` and `xir`. All instances of the class implement an accrue function with the type `x -> xir -> x`.

```
class Currency x xir where
  accrue :: x -> xir -> x
```

Classes are used for constraining the type parameters of generic code:

```
profitX :: Currency x xir => x -> xir -> x
profitX amount interestX = accrue amount interestX
```

An instance declaration defines the functions that satisfy a class' requirements for a particular set of type arguments.

```
newtype NOK = NOK Float
newtype NOKIR = NOKIR Float

instance Currency NOK NOKIR where
  accrue (NOK n) (NOKIR nir) = NOK (n * r)
```

The definitions of a type, class, and instance are all distinct: one can *retroactively* define an instance of a class for types that are already defined. This approach is more flexible than, say, Java, where the interfaces that a class implements are defined at the point of defining the class. On the other hand, this creates the possibility of overlapping and even conflicting instance declarations: two or more instance declarations match for specific argument types, and there may not be an ordering between the instance declarations that would resolve the conflict.

A.2 C++ Templates

C++ has supported a template mechanism since the first official 1998 standard. In the beginning C++ did not allow nested templates, i.e., a function in a templated class could not have additional template arguments. This was rectified a little later, and appeared in the next standard (2003).

Like macros, C++ templates are not typechecked before they are instantiated, but unlike macros they are typechecked when instantiated. The latter obviously is beneficial, while the former has significant drawbacks. Generally it is difficult to develop template code without support from the type system. Minor programming errors that are trivial to detect by the type system may easily go undetected. The only way to typecheck template code is to instantiate it with template arguments, but a developer's "limited" trial instantiations may be fine due to coincidences between the tests' template arguments and the template code expectations. This leaves the casual user of the template code with the dreaded instantiation errors, in extreme cases spitting out millions of lines of error messages relating to template code internals. These problems occur both in the case when there is an implementation error in the template code, or when the user instantiated the template with inappropriate template arguments.

Based on user needs (template typechecking, usefulness of instantiation error messages), the 2011 C++ standard was intended to include *concepts*, a mechanism for stating properties about template code, including axioms for template parameters. Concepts failed to be included due to conflicting design issues. A much scaled down version, *concept lite*, is now being attempted for C++ 2020. Concept lite allows the developer of template code to state expected properties for the actual template arguments, giving the unsuspecting user relevant feedback on instantiation errors. Unfortunately, the concept lite feature does not give the template code developer any help ensuring that the template code is compatible with the declared concept.

A.2.1 Template metaprogramming

The C++ templates mechanism has the property that it is a limited *compile-time metaprogramming* facility. Templates can be provided *specializations/overloads* to be used when the template is instantiated with a particular set of parameters, which works as a compile-time selection expression; together with recursive template definitions, compile-time computations can be expressed. Thus the type system acts as a de facto interpreter for a strange embedded programming language. Template metaprogramming achieved quite a lot of attention when discovered, and has been heavily used in libraries like BOOST, e.g., to achieve code optimisation in libraries. In itself

it is a bit weird and difficult to use feature that relies on clever language "hacks". Compile-time metaprogramming uses an unintuitive syntax, makes compilation slow, and increases a compiler's memory consumption considerably. Note that C++ template metaprogramming does not depend on the template mechanism's lack of internal typechecking. On the contrary, the lack of template code typechecking makes template metaprogramming even more difficult and error prone.

The C++ committee is working on *static reflection* as a more general approach to compile-time metaprogramming. For now, the *constexpr* feature allows for defining ordinary C++ functions to be interpreted at compile time (assuming of course their inputs are known at compile time too). More features are needed for full metaprogramming.

The C++ experience suggests that while compile-time computations can be useful for generic libraries, the mechanisms for compile-time computation and generics can and should be kept mostly orthogonal.

A.3 Java Generics

Issues with Java generics has been covered in two sections above.

The problem with not treating intrinsics and user defined types in the same way was discussed in Section 2.2.

The problem of using the inheritance hierarchy (interface inheritance) was discussed in Section 2.6.

References

[1] Bagge, A.H., David, V., Haveraaen, M.: Testing with axioms in C++ 2011. Journal of Object Technology 10, 10:1–32 (2011), https://doi.org/10.5381/jot.2011.10.1.a10

[2] Bagge, A.H., Haveraaen, M.: Axiom-based transformations: Optimisation and testing. In: Vinju, J.J., Johnstone, A. (eds.) Eighth Workshop on Language Descriptions, Tools and Applications (LDTA 2008). Electronic Notes in Theoretical Computer Science, vol. 238, pp. 17–33. Elsevier, Budapest, Hungary (2009)

[3] Bagge, A.H., Haveraaen, M.: Specification of generic APIs, or: Why algebraic may be better than pre/post. In: Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology. pp. 71–80. HILT '14, ACM, New York, NY, USA (2014)

[4] Bernardy, J.P., Jansson, P., Claessen, K.: Testing polymorphic properties. In: Gordon, A. (ed.) Programming Languages and Systems: Proceedings of the 19th European Symposium on Programming (ESOP 2010). Lecture Notes in Computer Science, vol. 6012, pp. 125–144. Springer-Verlag (2010), https://doi.org/10.1007/978-3-642-11957-6_8

[5] Burrows, E.: Programming with Explicit Dependencies: A Framework for Portable Parallel Programming. Ph.D. thesis, Research School in Information and Communication Technology, Department of Informatics, University of Bergen, Norway, PB 7803, 5020 Bergen, Norway (2011), http://bldl.ii.uib.no/phd/burrows.pdf

[6] Haveraaen, M.: Proving a core code for FDM correct by 2 + dw tests. In: Scholz, S., Shivers, O. (eds.) Proceedings of the 5th ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming, ARRAY@PLDI 2018, Philadelphia, PA, USA, June 19, 2018. pp. 42–49. ACM (2018), http://doi.acm.org/10.1145/3219753.3219759

[7] Mosses, P.D. (ed.): CASL Reference Manual: The Complete Documentation of the Common Algebraic Specification Language, Lecture Notes in Computer Science, vol. 2960. Springer-Verlag (2004), https://doi.org/10.1007/b96103

[8] Stepanov, A., McJones, P.: Elements of Programming. Addison-Wesley Professional, 1st edn. (2009)

[9] Stroustrup, B., Sutter, H.: Cpp core guidelines. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines (2019)

[10] Wadler, P.: Theorems for free! In: Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture. pp. 347–359. FPCA '89, ACM, New York, NY, USA (1989), http://doi.acm.org/10.1145/99370.99404