Exploring the boundaries of Ada syntax with functional-style iterators

Alejandro R. Mosteo[1,2], María-Teresa Lorente[3,]

[1] Instituto de Investigación en Ingeniería de Aragón (I3A)
`amosteo@unizar.es`,
C/ Mariano Esquillor s/n, 50018 Zaragoza, Spain
[2] Centro Universitario de la Defensa de Zaragoza (CUD)
Ctra. de Huesca s/n, 50090 Zaragoza, Spain
[3] Natural Robotics Lab, The University of Sheffield
Pam Liversidge Building, Sheffield S1 3JD, United Kingdom

Abstract. Functional-style iterators are present in many popular languages as a way of processing data in several connected steps, in a safe and readable manner. By contrast, Ada started providing general iterators only with its 2012 revision, in a way not directly intended for sequential composition. This paper presents a functional iterators library design inspired by the standard Rust iterators and the RxAda reactive extensions. This library is used as a case study for the limitations in implementing higher-kinded types in current Ada, and how currently proposed extensions for the 202X revision might change the situation.

Keywords: Iterator Pattern, Pull Framework, Containers, Ada 2012
Final version: `https://doi.org/10.1016/j.sysarc.2021.102039`

1 Introduction

Item collections are at the core of many computer applications. These collections may store widgets in a graphical user interface, entities in a game, structured data in documents, and so on. Traversing containers and manipulating its contents is thus a typical activity that programming languages support in several ways. Furthermore, item processing and transformation via data chains is a recognized design pattern [19], also in high-integrity domains [7]. For this reason, some languages provide convenient syntax to naturally express such chaining constructions [10], and whole frameworks build on these ideas [13].

Ada containers were not standardized until the 2005 language revision [4]. This initial version relied on cursors for manual iteration and provided a single `Iterate` procedure [4] for complete iteration akin to a `for each loop`, which in turn takes a procedure access argument, inverting the presentation flow. These facilities, although effective, resulted in cumbersome nested constructions for even simple operations, or in explicit cursor management, which negates the range safety existing in regular array iteration using the `'Range` attribute.

[4] Ada Reference Manual 2005 A.18.2 73/2

These limitations were recognized and addressed through the new `"of"` syntax and iterators [1] in Ada 2012. The use of appropriate iteration aspects moved the boilerplate from the user of an iterable entity to the provider of the collection, recovering the regular concise and safe syntax of Ada loops for the use with the standard containers and any other iterable constructs.

Such boilerplate, though, is still the subject of improvements with recent proposals [14] aimed at providing simpler definition of iterators. Still, current and proposed Ada iterator concepts are not intended for composition, leaving room for further contributions in this direction. In the same way that Reactive Extensions [9,13] can be considered a generalization of the observer pattern, functional iterators generalize the plain iterator to allow the composition of sequential processing pipelines, bringing the benefits of range safety and clarity of expression to multi-step transformations of the items in a collection.

We set out to make two contributions with this work. Firstly, we present a functional iterators library [12] for the current language revision, bringing Ada closer to other languages in this regard. Secondly, we identify and discuss difficulties found with current Ada syntax, and we review current proposals for Ada 202X to identify their impact in improving expressiveness in these kinds of problems.

The paper is structured as follows: Section 2 introduces with examples the basics of functional iterators as found in other languages. Section 3 presents our implementation design and main features. Next, Section 4 discusses the limitations we found in current Ada syntax and reviews upcoming changes and proposals that could simplify expressiveness. Lastly, concluding remarks and future directions close the paper in Section 5.

2 Functional Iterators Overview

Plain iterators, as usually found in many languages, can be boiled down to the following Ada code:

Listing 2.1. Minimal iterator definition

```ada
generic
   type Any_Element (<>) is private;
package Iterators is

   type Iterator is interface;

   function Has_Next (This : Iterator) return Boolean is abstract;
   --  Check if iteration has ended

   function Next (This : in out Iterator) return Any_Element is abstract;
   --  Retrieve next element in the sequence

end Iterators;
```

Strictly functional languages like Haskell are characterized by purity of function calls and encapsulation of side-effects via constructs like monads [18]. In this work we are referring to a looser concept of functional style. By functional iterators we mean the ability to chain calls in such a way that iteration safety over the iterable is preserved, whereas state is encapsulated in each iterator. This iteration style can be found in several currently popular languages that do not claim to be primarily functional, like C#, Java, JavaScript, Rust, etc. Consequently, the "functional-style" moniker implies that our iterators are not actually functional, as is true for Ada as well. This is evidenced by the subprogram `Next` in Listing 2.1 having side-effects. Instead, the term functional is loosely used to imply that iterators can be chained together while possibly encapsulating a state that is not required externally:

Listing 2.2. Example of chaining iterator operators

```
type Operator is new Iterator with private;
--  For now, it suffices to say that an Operator is in itself an Iterator

function "&" (LHS  : Iterator'Class; -- Upstream iterator
              RHS  : Operator'Class) -- Inserted operation
              return Iterator'Class; -- Downstream iterator view of the inserted operator

declare
   Targets : constant Iterator'Class :=
      Some_Collection.Iterate        -- Provides an iterator
      & Filter (<some condition>)    -- Takes an iterator and returns another one
                                     -- Lets pass only certain elements
      & Take (<number>)              -- Limit iteration to at most <number> elements
      & Map (<some transformation>)  -- Applies a function to each element
```

In the example in Listing 2.2, the crux of the matter is the `"&"` function, which takes a left-hand iterator and right-hand operator, assembling them and returning in turn a new iterator. Typically, libraries with this kind of chainable construct also provide a suite of operators for general needs related to filtering, reduction, and mapping [17].

Operators can be intermediate or terminal. An assembled sequence of intermediate operators is passive/lazy (no iteration is triggered), so we still need to write something that calls `Next` on the resulting sequence to go over the elements, or that uses a `for Element of Iterator_Value loop` to traverse all the elements. (Terminal operators also trigger iteration, hence the name.) For example,[5] Listings 2.3 and 2.4 show side-by-side the computation of the average of values in a vector of floats; `Reduce` in the iterator-based Listing 2.4 is a terminal, which consumes the iterator sequence in its entirety.

Listing 2.3. Loop-based reduction

```
declare
   Average : Float := 0.0;
begin -- Data is an array of Float
   for I in Data'Range loop
      Average := Average + Data (I);
   end loop;
   Average := Average / Float (Data'Length);
```

Listing 2.4. Iterator-based reduction

```
declare
   Average : constant Float :=
     (Const_Iter (Data) -- Iterator from array
      & Reduce (0.0, "+"'Access))
        --  Terminal Reduce consumes the
        --  iterator and returns a value.
     / Float (Data'Length);
```

[5] All code examples are available and compilable at `https://github.com/mosteo/iterators/blob/JSA-2020/src/iterators-demo-jsa_20.adb`

Since elements have to be “pulled” using the `Next` call (although this is rarely done explicitly), this kind of functional operator is considered a pull framework. By contrast, push frameworks like ReactiveX [8,11] push down elements onto subscribers, allowing one-to-many data flows. Also, while ReactiveX is a much more ambitious framework, with tasking capabilities, functional iterators are typically intended for use as regular localized loops.

While, in essence, we are sugaring plain loops, in conjunction with the `Chain` and monad-inspired `Flat_Map` operators (Listing 2.5), we gain a more general framework in which looping actions can be composed via corresponding iterators. Iterable objects can be combined through `Flat_Map` to obtain a new iterator that provides a single sequence of elements. A complete example for a realistic computation involving these concepts is later presented, in Listing 3.5.

Listing 2.5. The `Flat_Map` operator

```
function Chain (Next : Iterator'Class) return Operator'Class;
--  Take all elements from the upstream iterator, and then take the elements from Next iterator

function Flat_Map (Map : access function (E : Any_Element) return Iterator'Class)
                  return Operator'Class;
--  Applies Map to every element taken from the upstream iterator.
--  Take in turn the elements from the generated iterators.
```

Finally, as long as we use only filtering operations, elements can be iterated over by reference. The following examples in Listings 2.6 and 2.7 show side-by-side how the same operation of modifying in place some elements in a collection could be done in the traditional way and with the new iterators. Iterators can be looped over with normal `"of"` notation, or using a `For_Each` call that applies a procedure to all elements in a sequence. In this example we are marking as ‘winners’ at most the first ten qualified elements.

Listing 2.6. Plain loop

```
declare
   Count : Natural := 0;
begin
   for Element of Collection loop
      if Is_Candidate (Element) then
         Count := Count + 1;
         Set_Winner (Element);
      end if;
      exit when Count = 10;
   end loop;
end;
```

Listing 2.7. Iterator loop

```
--  Iterator'Class supports "of" syntax.
--  Alternatively, For_Each can be used.
For_Each
   (Iter (Collection)
       --  Get read/write iterator
    & Filter (Is_Candidate'Access)
       --  Is_Candidate must return boolean
    & Take (10),
       --  Stops after at most ten elements
    Set_Winner'Access);
       --  Performs closure on each element
```

Although it is arguable which style is clearer, nowadays many programmers are familiar with functional-style chaining of actions. Operators can also encapsulate state, removing it from the surrounding code, as the counter `Count` in the previous examples shows. Other possibilities include more explicit conditions on collections:

Listing 2.8. Using iterators in contracts

```
procedure Complex_Algorithm (Collection : in out Any_Collection)
```

```
   with Pre => Iter (Collection)
               & Filter (Condition'Access)
               & Reduce (Computation'Access) = ...;
```

3 Library Design

The examples in the previous section deliberately hid syntactic difficulties with the implementation. In this section, such details are exposed during the design descriptions, to facilitate discussion in Section 4. As in the RxAda library [11,13], compile-time type safety is a cornerstone of the design, in turn being the reason for most syntactic hurdles. First we present the main classes in the library, to follow with available operators and conclude the section with practical use observations.

3.1 Main Classes

The Iterators library relies on only a handful of types, that are presented in Listings 3.1 and 3.2. At the root is the `Iterator` class, whose `Next` call returns a `Cursor` (instead of directly returning an element, as simplified in Section 2). The use of a cursor is mandated to enable standard Ada `"of"` notation with the new iterators. This imposition, however, serves to also provide read-only and read-write iteration. Whereas Ada uses the object after `"of"` to decide if the loop variable is writable or not, in functional iterators we have the option to request a writable iterator (assuming the base collection is also writable), that will allow by-reference modification of elements, or a constant iterator that could be relayed ensuring immutability of the underlying collection. By encapsulating in the cursor the writable property, duplication of iterator implementation is avoided in derived operators.

Listing 3.1. Basic types in the Iterators library

```
generic
   type Any_Element (<>) is private;
package Iterators.Root is -- All of the library packages are under the umbrella of Iterators

   --  Before defining the Iterator proper, we need to define the cursor that will be used for
         standard "of" syntax:

   type Cursor is tagged private;
   function Has_Element (This : Cursor) return Boolean;
   function Is_Empty (This : Cursor) return Boolean;

   function Element (This : Cursor) return Any_Element;
   --  Get a copy of the element
   function Get (This : Cursor) return Const_Ref;
   --  Get a read-only reference to the element
   function Ref (This : Cursor) return Reference;
   --  Get a read-write reference to the element

   function New_Cursor       (Element : aliased in out Any_Element) return Cursor;
   function New_Const_Cursor (Element : aliased        Any_Element) return Cursor;
   --  Helpers to construct a cursor from an element. The use of a read-only or read-write cursor at
         the beginning of a chain will enforce proper usage at iteration time in downstream clients.
   package Ada_Iterator_Interfaces is new Ada.Iterator_Interfaces (Cursor, Has_Element);
   --  Needed for standard "of" notation

   --  Proper iterator class with standard "of" iteration aspects:

   type Iterator is interface with
     Constant_Indexing => ...,
     Variable_Indexing => ...,
     Default_Iterator  => ...,
     Iterator_Element  => Any_Element;
   --  Particulars of the standard Ada iteration aspects are elided, as they are not
   --  needed for the discussions in this section.

   function Next (This : in out Iterator) return Cursor'Class is abstract;
   --  Get next element, if any, through a Cursor

   --  Other declarations omitted. Please consult the online repository for full code.

end Iterators.Root;
```

3.2 Operators and other chainables

Operators (in Listing 3.2) are the specialization of iterators that enable chaining. In the general case, an operator takes an element of some upstream type and converts it to another downstream type. This has implications on how to chain operators, addressed in Subsection 3.4. For a given operator in a chain, its *upstream* iterator is the one immediately adjacent towards the source of elements, and analogously for *downstream*.

Internally, operators hold their direct upstream iterator, to be able to call `Next` on it. Operators that do not transform between types (e.g., filters) are a specialization that can pass along references to elements, preserving by-reference semantics, as was shown in Listing 2.7.

Listing 3.2. The Operator type

```
generic
   with package From is new Root (<>); -- Iterables of upstream type   (to pull from)
   with package Into is new Root (<>); -- Iterables of downstream type (implemented here)
package Iterators.Operators is

   type Operator is abstract new Into.Iterator with private;

   function "&" (LHS :  From.Iterator'Class;
                 RHS :  Operator'Class)
                 return Into.Iterator'Class;
   --  Real signature of the "&" function. An upstream iterator is stored in the operator, that will
   --  use it to pull an upstream element via Next. The concrete operator implementation will have to
   --  transform that element into a downstream element as part of its duties. See Map and Flat_Map
   --  below as an example. The operator is returned as a downstream iterator, so it can become the
   --  left-hand operand of further chaining.

   function Flat_Map (Map : not null access
                        function (E : From.Any_Element) return Into.Iterator'Class)
                      return Operator'Class;
   --  While upstream provides elements, generate new iterators by calling Map on them.
   --  Take elements from these new iterators and pass them downstream.
   --  Other signatures exist based on variations of the idea of iterating over iterators.

   function Map (Map : not null access
                 function (E : From.Any_Element) return Into.Any_Element)
               return Operator'Class;
   --  Pull one element, convert it by calling Map on it, and provide the result to downstream.

   --  More declarations omitted for clarity here.

end Iterators.Operators;
```

A few special iterators are also defined, called `Sources`. These are functions that construct an Iterator and intend to serve as the root of a chain. Trivial examples are `Empty`, that always ends without elements, and `Just`, that converts a single element into an iterator (sometimes called `Unit` in other languages).

Generator packages exist for Ada arrays and standard containers (see Listing 3.3 for an example). Traits packages [5] are defined to abstract away necessary subprograms of a particular container (they could also be used with custom containers) and are used as package generic formals in the library (see, for example, the second generic formal in Listing 3.3). Specialized traits for keyed containers (vectors, maps), where an index into elements can be also of interest, allow the coupling of elements and their key into a `Pair`, for which iterators can also be obtained.

Listing 3.3. Generators for containers via traits

```
generic
   with package Root is new Standard.Root (<>);
   with package Container_Traits is new Traits.Containers (Any_Element => Root.Any_Element,
                                                         others      => <>);
   --  The element type provided by the iterator and the one stored in the container must match.
package Iterators.Generators.Containers is

   function Const_Iter (Container : aliased Containers.Container) return Root.Iterator'Class;
   --  Read-only iterator

   function Iter (Container : aliased in out Containers.Container) return Root.Iterator'Class;
   --  Read-write iterator

   --  More declarations omitted for clarity here.

end Iterators.Generators.Containers;
```

At the other end of the chain exist the `Collectors`. As a minimum, each iterator package provides a plain list type and `Collect` operator, that causes the chain to be iterated over and the final elements to be collected in such a list (that could also be used in turn as the source for further iteration). Also, the same traits packages used to turn standard containers into iterators allow the instantiation of collector operators for standard containers.

3.3 Instantiation

To be able to use these iterators, a number of package instantiations have to be made. To jump-start the use of iterators for a concrete type, the generic package `Iterators.Elements` bundles the instantiation of the root Iterator class for the element type, the type-preserving operators, iterators that generate iterators over iterators (here called metaiterators), and a few convenience functions (Listing 3.4).

Listing 3.4. The `Iterators.Elements` package.

```
--  With clauses omitted.
generic
   type Any_Element (<>) is private;
package Iterators.Elements with Preelaborate is

   --  Defines the Iterator'Class for the element type.
   package Iterators is new Standard.Iterators.Root (Any_Element);
   package Iters renames Iterators;

   --  Defines the type-preserving Operator'Class for the type, and operators of that class.
   --  These include intermediates like Filter and terminals like Collect, Reduce.
   package Operators is new Iterators.Operators;
   package Op renames Operators;

   --  Defines specialized operators that group/ungroup elements into iterators.
   --  These include the Flat_Map, Window used in later examples.
   package Meta_Operators is new Standard.Iterators.Meta (Iterators, Operators);
   package Meta renames Meta_Operators;

   --  More declarations omitted for clarity here.

end Iterators.Elements;
```

Listing 3.5 shows a concrete example involving a couple of types in both directions of transformation. The example computes and prints the running average of floating point values stored in a file, with possibly invalid samples to be filtered out. Running averages appear in many applications as low-pass filtering, financial indicators, etc. The example relies on the `Window` operator to group values into new iterators that, in turn, are flattened back into the main sequence by `Flat_Map`, which computes the sum of each windowed sequence of elements (Scan takes an initial value and applies a function to each incoming value and the previous result). For comparison, equivalent traditional (without using Ada iterators) and Ada 202X (using Ada iterators and planned features) implementations are provided in Appendices A and B.

The root iterators for a type have to be instantiated first. This package serves as a generic formal for type-transforming operators. Supplementarily, if specific generators or collectors are needed, these can also be instantiated using the root iterator package.

Listing 3.5. Example of instantiation and use.

```
procedure Print_Running_Average (File_Name : String; Width : Positive := 3) is
   --  Compute the running average of float values found in a file, one per line, with -1.0 meaning
         an invalid value to be skipped. Width indicates how many consecutive values are taken to
         generate an averaged value. Example: 3.0, 4.0, 5.0, 6.0, 5.0 produces 4.0, 5.0, 5.3.

   use Iterators; -- All packages in the library are under the Iterators package.

   --  Instantiations to iterate on Floats and convert from/to Strings. String iterators are
         preinstantiated by the library in package Iterators.Std.Strings, so they are not
         instantiated here again.
   package Float_Iters is new Elements (Float); -- Iterators and Operators for Floats.
   package Float2Str   is new Operators (Float_Iters.Iterators, Std.Strings.Iterators);
   package Str2Float   is new Operators (Std.Strings.Iterators, Float_Iters.Iterators);

   --  Mandatory "uses" for "&" visibility.
   use Float_Iters.Linking;      -- Plain Float operators linking.
   use Float_Iters.Meta.Linking; -- Float from/to Float_Iterable operator linking.
   use Float2Str.Linking;        -- Float to String operators linking.
   use Str2Float.Linking;        -- String to Float operators linking.

   --  Optional "uses", to make iterator operators directly visible without prefixing. The prefixes
         are shown nonetheless in the actual sequence below, for clarity of operator provenance.
   use Float_Iters.Meta; -- Operators that generate/consume iterators over Float iterators.
   use Float_Iters.Op;   -- Operators that apply to the Float element type.
   use Float2Str;        -- Operators that transform from Float into String.
   use Std.Strings.Op;   -- Operators that apply to the String element type.
   use Str2Float;        -- Operators that transform from String into Float.

   --  Supporting functions that would be typically provided as lambdas in languages that have them.
   function Div      (F : Float) return Float   is (F / Float (Width));
   function Is_Valid (F : Float) return Boolean is (F >= 0.0);

begin
   -- ENCLOSING PACKAGE   -- OPERATOR/SUBPROGRAM    -- EXPLANATION
   Std.Strings.Op         .For_Each                 -- Process the complete sequence.
     (Text_IO             .Lines (File_Name)        -- String iterator that reads lines from a file.
      & Str2Float         .Map (Value'Access)       -- Convert to Float.
      & Float_Iters.Op    .Filter (Is_Valid'Access) -- Skip negative numbers (invalid samples).
      & Float_Iters.Meta  .Window (Size => Width,   -- Group <Size> elements into a new iterator
                                  Skip => 1)        --   and advance <Skip> elements.
      & Float_Iters.Meta  .Flat_Map                 -- Iterate over each Window-generated iterator.
         (Float_Iters.Op  .Scan (0.0, "+"'Access)   -- Apply "+" to prev result and incoming value.
          & Float_Iters.Op.Last)                    -- Take last value only (sum of each Window).
      & Float_Iters.Op    .Map (Div'Access)         -- Divide by window size.
      & Float_Iters.Op    .Map (Image'Access),      -- Back to string.
      GNAT.IO             .Put_Line'Access);        -- Print each running average value.
end Print_Running_Average;
```

3.4 Chain syntax

The linking of iterators and operators to build chains poses a specific challenge in Ada. Normally, in other languages, operators are accessed through dot notation (e.g., in Java, where generics mixing several types are no obstacle, or in Rust, via traits, or in general in any language supporting Uniform Function Call Syntax (UFCS) [10]). Listings 3.6 and 3.7 show the prototypes of the `Map` operator in Java [6] and Rust [15] standard libraries. Ada only supports dot notation for primitive operations of tagged types. Since operators may transform across iterator types (coming from different instances), it is impossible to have primitive functions returning values of types declared in different instances of the same generic. In other words, a generic cannot reference itself, nor two different generics can refer to each other in Ada, unlike in the Java or Rust examples. Because of this characteristic, we were unable to find a complete solution using dot notation only. Making the operators primitive would have been our preferred solution, because it would eliminate the need for `use` clauses in clients of our library, but this, alas, proved to be impossible.

Listing 3.6. Java Map operator

```
<R> Stream<R>
  map(Function<? super T,? extends R> mapper)
// T is the upstream element type
// R is the downstream element type
```

Listing 3.7. Rust Map operator

```
fn map<B, F>(self, f: F) -> Map<Self, F>
where F: FnMut(Self::Item) -> B
// Self::Item is the upstream element type
// B is the downstream element type
```

We first identified this circumstance during development of RxAda, and we found that an alternative in Ada is to use a standard operator function overloading, such as `"&"`, which furthermore conveys the idea of concatenation to Ada programmers. The operator is able to bridge the gap between two instances of operators (Listing 3.2) because a particular `Iterator` class can appear as both an `Into.Iterator’Class` generic formal in one operator instantiation and a `From.Iterator’Class` in another instantiation.

The drawback of this approach is that it requires to `use` every instance (or type) where the operator is defined. While in RxAda there is one instantiation per type and transformation, in the library being presented we may need more, for operators that cause collection or reduction, since these return their own final types and have different signatures.

Given this inconvenience, an imperative alternative (Listing 3.8) is experimentally provided, that comes at the price of having to declare a `Chain` (an iterator holder with proxying of the operators) for every element type and transformation, but in return making `use` clauses unnecessary, and having all operators as primitive calls.

This alternative is not satisfactory either, since it merely translates the need for `use` clauses into operator holders, requiring error-prone manual housekeeping of chaining and resulting in less readable code.

Listing 3.8. Example of imperative syntax

```ada
procedure Print_Running_Average_Imperative (File_Name : String; Width : Positive := 3) is
   --  Same instantiations omitted.

   S2F : Str2Float.Chain;
   Flt : Float_Iters.Operators.Chain;
   F2M : Float_Iters.Meta.Chain_To_Meta;
   M2F : Float_Iters.Meta.Chain_From_Meta;
   Sum : Float_Iters.Operators.Chain;
   Avg : Float_Iters.Operators.Chain;
   F2S : Float2Str.Chain;
begin
   --  Chain up to Window operator.
   S2F.Start  (Text_IO.Lines (File_Name));
   S2F.Map    (Value’Access);
   Flt.Resume (S2F);
   Flt.Filter (Is_Positive’Access);
   F2M.Resume (Flt);
   F2M.Window (Size => Width, Skip => 1);
   M2F.Resume (F2M);

   --  Subchain for aggregation with Flat_Map.
   Sum.Start    (Float_Iters.Op.Scan (0.0, "+"’Access));
   Sum.Continue (Float_Iters.Op.Last);
   M2F.Flat_Map (Sum.As_Iterator);

   --  Final chain up to printing.
   Avg.Resume (M2F);
   Avg.Map    (Div’Access);
   F2S.Resume (Avg);
   F2S.Map    (Image’Access);

   F2S.For_Each (GNAT.IO.Put_Line’Access);
end Print_Running_Average_Imperative;
```

A mixed solution, using dot notation for type-preserving operators (hence staying within the same iterator instantiation), and `"&"` for type-transforming operators would also work, but is unsatisfactory because of the lack of notation uniformity. This mixed approach would also entail associativity issues, since `"&"` connections would be applied after dot calls.

The bottom line is that we were unable to find a complete solution that does not require sacrifices using current Ada syntax. As a result, the barrier to opportunistically use the library is raised given that it feels heavyweight, with many instantiations and `use` clauses even for relatively simple chains, and poorly integrated with the language. For this reason, we studied amendments and proposals for Ada 202X in the hope of finding any improvements. These amendments that are related in some way to this library are the subject of Section 4.

4 Related Ada 202X Amendments and Proposals

In this section we discuss several proposals for Ada enhancement, some of them already accepted and completed, and some in earlier stages, that could improve the ergonomics of our library, to conclude with our view on the main remaining challenges.

4.1 Generalized dot notation

At first glance, the most promising proposal is found in the AdaCore repository for Ada and Spark Requests For Comments [16]. The proposal named "prefix untagged" addresses extending dot notation to untagged types, so it might conceivably have contained some form of UFCS for non-primitive subprograms. On closer inspection this is not the case, as the distinction between primitive and non-primitive subprograms is still integral to the proposal. Since generic instantiations created after the root `Iterator` class for a type can never provide new primitive subprograms, this proposal does not change the current *status quo.*

4.2 Implicit instantiations

Explicit instantiations are a difference between Ada and most comparable languages (C++, Java, Rust, for example). The need to create all instantiations in advance leads to a certain amount of boilerplate, whose justification we are not going to address. As the AI12-0215-2 [2] on implicit instantiations evidences, at least in some cases it is considered to cause a worsening of the signal-to-noise ratio.

We showed before that the use of our library requires instantiations for every element type and pair of types involved in a transformation. Were this AI (unfinished at the time of this writing) to be adopted, a reduction of the number of instantiations could happen. The first observation is that the use of the `"&"` operator is ruled out, since we need to refer to the containing instance to be able to "use" it (in the Ada sense). By the same reasoning, operators that transform across types would still need explicit instantiations of the root iterators of each type, since they take other packages as generic formals.

Only in the case of single-type operators would this AI lead to the elimination of the single instance for a type, allowing the declaration of an imperative-style `Chain` variable without a prior explicit instantiation.

4.3 Anonymous bodies for closure iterators

The already completed AI12-0189-1 amendment [3], "loop-body as anonymous procedure", will repurpose the Ada 2005 closure-style iterator procedures in the standard containers to allow simpler writing of loops. In this proposal, the arguments of the closure call act as a tuple that becomes the index of the loop, while an anonymous body is created behind the scenes, containing the body of the loop, to act as the closure parameter to the Iterate container call (see Listing 4.1). This feature will be usable with our library iterators for keyed containers, that already produce a pair with key and element for iteration. Boilerplate to extract the key and value from the cursor will be eliminated by this AI.

Listing 4.1. Explicit closure

```
declare
   procedure Process (Key : String;
                      Val : Any_Element) is ...;
begin
   Some_Container.Iterate (Process'Access);
```

Listing 4.2. Anonymous procedure

```
for (Key, Val) of Some_Container.Iterate loop
   --  Same code as in Process body in left
       example
end loop;
```

4.4 Towards a complete solution

A common saying in programming is that you have to be "idiomatic"; that is, to use each language particular strengths, intended constructions, and avoid to, e.g., "program C in Ada". In the case of our RxAda and Iterators libraries, that could imply that we are simply trying to coerce Ada into a functional style that is out of its scope. Each reader may lend a different weight to this argument. However, in our view, what prevents an entirely satisfying solution is more a case of an unfortunate compounding of syntax rules in the language than a deliberate attempt at staying imperative at the expense of functional style. Our evaluation points to an unwanted consequence (the inability to conveniently implement some semblance of higher-kinded types, or interrelated families of generics) that we have not seen addressed before.

We have not yet found a viable proposal to fill this gap, although we have narrowed down the factors at play to the restrictions on dot notation and inability of a generic to refer to another instance of itself. We feel that a solution based on tweaking or extending any of these aspects (perhaps, indeed, defining new Ada aspects, or allowing a limited view of a generic within itself) can be worked out. We consider this an open line of future work, with difficult interrelated aspects to consider.

5 Conclusions

This paper presented an Ada 2012 implementation of functional-style iterators, describing its main design ideas. The library is intended to be both compatible with the standard Ada containers and adaptable to other third-party container libraries. Key features of the library are its composability to allow processing of elements through several operators and by-reference element semantics, where possible, to minimize overhead and preserve the ability to modify elements in-place.

The library serves as a basis for discussion of Ada syntax peculiarities that prevent the adoption of the usual dot notation with total regularity. This led us in turn to review possible improvements with incoming 202X language extensions: The completed proposal on anonymous procedures for loop bodies will simplify the use of iterators for keyed containers. Anonymous implicit generics will reduce the boilerplate currently needed to use the library in some particular cases only. The proposed extension of dot notation to untagged types will not change the issue of generic families being unable to reference sibling instances with primitive operations, which is in our opinion the biggest hurdle still to be surmounted.

The Iterators library is available under an Open Source license to interested parties [12].

Aknowledgements

This work has been supported by the Spanish projects DPI2016-76676-R-AEI/FEDER-UE, PID2019-105390RB-I00, CUD2019-05, CUD2020-14, and DGA-T45_20R.

References

1. Ada Rapporteur Group: AI05-0139-2: Syntactic sugar for accessors, containers, and iterators (2011), `http://www.ada-auth.org/cgi-bin/cvsweb.cgi/ai05s/ai05-0139-2.txt?rev=1.22&raw=N`
2. Ada Rapporteur Group: AI12-0215-2: Implicit instantiations (2011), `http://www.ada-auth.org/cgi-bin/cvsweb.cgi/ai12s/ai12-0215-2.txt?rev=1.1&raw=N`
3. Ada Rapporteur Group: AI12-0189-1: Loop-body as anonymous procedure (2018), `http://www.ada-auth.org/cgi-bin/cvsweb.cgi/ai12s/ai12-0189-1.txt?rev=1.14&raw=N`
4. Barnes, J.G.P.: Rationale for Ada 2005. AdaCore (2006)
5. Briot, E.: Traits-based containers (2015), `http://blog.adacore.com/traits-based-containers`
6. Java 8 API: Java stream interface (2014), `https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html`
7. Kloda, T., Bertout, A., Sorel, Y.: Latency upper bound for data chains of real-time periodic tasks. Journal of Systems Architecture 109, 101824 (2020), `http://www.sciencedirect.com/science/article/pii/S1383762120301168`
8. Maglie, A.: ReactiveX and RxJava. In: Reactive Java Programming, pp. 1–9. Springer (2016)

9. Meijer, E.: Subject/observer is dual to iterator. In: FIT: Fun Ideas and Thoughts at the Conference on Programming Language Design and Implementation (2010)
10. Michael, P.: Origins of the D programming language. In: ACM SIGPLAN history of programming languages conference 2020 (2018)
11. Mosteo, A.R.: RxAda: An Ada implementation of the ReactiveX API. In: Ada-Europe International Conference on Reliable Software Technologies. pp. 153–166. Springer (2017)
12. Mosteo, A.R.: Ada 2012 functional iterators (2019), `https://github.com/mosteo/iterators`
13. Mosteo, A.R.: Reactive programming in Ada 2012 with RxAda. Journal of Systems Architecture 110, 101784 (2020), `http://www.sciencedirect.com/science/article/pii/S1383762120300783`
14. Reznik, M.: Lightweight iterators (2019), `https://github.com/AdaCore/ada-spark-rfcs/pull/37`
15. Rust API: Rust iterator interface (2017), `https://doc.rust-lang.org/std/iter/trait.Iterator.html#method.map`
16. Taft, T.: Ada RFC, prefix untagged (2019), `https://github.com/AdaCore/ada-spark-rfcs/pull/34`
17. Urma, R.G., Fusco, M., Mycroft, A.: Java 8 in Action: Lambdas, Streams, and functional-style programming. Manning Publications Co. (2014)
18. Wadler, P.: Monads for functional programming. In: International School on Advanced Functional Programming. pp. 24–52. Springer (1995)
19. Šljivo, I., Uriagereka, G.J., Puri, S., Gallina, B.: Guiding assurance of architectural design patterns for critical applications. Journal of Systems Architecture 110, 101765 (2020), `http://www.sciencedirect.com/science/article/pii/S138376212030059X`

A Classical implementation of running average example

The following Listing A.1 presents a functionally equivalent implementation of the iterators-based Listing 3.5, without using the Iterators library, and using only standard Ada features without defining any additional Ada iterators. Compilable source code for all versions of the example can be found at `https://github.com/mosteo/iterators/blob/JSA-2020/src/iterators-demo-jsa_20.adb`.

Listing A.1. Computing a running average without functional iterators

```
procedure Print_Running_Average_Classical (File_Name : String; Width : Positive := 3) is
   package Lists is new Ada.Containers.Doubly_Linked_Lists (Float);
   use Ada.Text_IO;
   File   : File_Type;
   Window : Lists.List;
begin
   Open (File, In_File, File_Name);

   while not End_Of_File (File) loop
      declare
         Line : constant String := Get_Line (File);
         Num  : constant Float  := Float'Value (Line);
      begin
         --  Skip negative numbers, used to signal invalid samples
         if Num >= 0.0 then

            --  Add new sample
            Window.Append (Num);

            --  Prune the window
            if Natural (Window.Length) > Width then
               Window.Delete_First;
            end if;

            --  New averaged value?
            if Natural (Window.Length) = Width then
               declare
                  Total : Float := 0.0;
               begin
                  --  Compute window total. This could be done more efficiently by keeping a running
                        total. However, that might have numerical implications due to unneeded
                        subtractions and would not be exactly equivalent to the Iterator version.
                  for Sample of Window loop
                     Total := Total + Sample;
                  end loop;

                  --  Write the new running average value
                  Put_Line (Float'Image (Total / Float (Width)));
               end;
            end if;
         end if;
      end;
   end loop;

   Close (File);
end Print_Running_Average_Classical;
```

B Ada 202X implementation of running average example

The following Listing B.1 presents a functionally equivalent implementation of the iterators-based Listing 3.5, using planned features from the future Ada 202X standard (loop filters and reductions), and defining an additional Ada iterator that provides lines of text files. There are no available compilers yet able to build this example. Note also that the use of the Ada iterator, although possible with existing Ada features, requires non-negligible extra code as the Ada Standard Library does not provide iterators over files. (See Listing B.2 for the necessary public declarations; private declarations and bodies can be found on the on-line repository.)

Listing B.1. Computing a running average using Ada 202X features.

```
procedure Print_Running_Average_Ada202x (File_Name : String; Width : Positive := 3) is
   package Lists is new Ada.Containers.Doubly_Linked_Lists (Float);
   use Ada.Text_IO;
   Window : Lists.List;
begin
   for Line of Lines ("file_avg_demo.txt") when Float'Value (Line) >= 0 loop
      Window.Append (Float'Value (Line));
      --  Prune the window
      if Natural (Window.Length) > Width then
         Window.Delete_First;
      end if;
      --  New averaged value?
      if Natural (Window.Length) = Width then
         GNAT.IO.Put_Line -- Compute and print new running average value.
           (Float'Image (Window'Reduce ("+", 0.0) / Float (Width)));
      end if;
   end loop;
end Print_Running_Average_Ada202x;
```

Listing B.2. Definitions for an Ada iterator over the lines of a text file.

```
type Ada_Cursor is limited private;
function Has_Element (Cursor : Ada_Cursor) return Boolean;

package Ada_Iterator_Interfaces is new Ada.Iterator_Interfaces (Ada_Cursor, Has_Element);

type Ada_Iterator (<>) is new Ada_Iterator_Interfaces.Forward_Iterator with private with
  Constant_Indexing => Element, -- Read-only element retrieval.
  Default_Iterator  => Iterate, -- Forward iteration.
  Iterator_Element  => String;  -- Returned type during iteration.

function Lines (File_Name : String) return Ada_Iterator;
--  Creates the Ada iterator.

function Element (This : aliased Ada_Iterator'Class; Pos : Ada_Cursor) return String;

function Iterate (This : aliased Ada_Iterator) return Ada_Iterator_Interfaces.Forward_Iterator'Class;

overriding function First (This : Ada_Iterator) return Ada_Cursor;

overriding function Next (This : Ada_Iterator; Position : Ada_Cursor) return Ada_Cursor;
```