Internal Iteration Externalized

Thomas Kühne

Staffordshire University, UK
T.Kuehne@soc.staffs.ac.uk

Abstract. Although it is acknowledged that internal iterators are easier and safer to use than conventional external iterators, it is commonly assumed that they are not applicable in languages without builtin support for closures and that they are less flexible than external iterators.
We present an iteration framework that uses objects to emulate closures, separates structure exploration and data consumption, and generalizes on folding, thereby invalidating both the above statements. Our proposed "transfold" scheme allows processing one or more data structures simultaneously without exposing structure representations and without writing explicit loops.
We show that the use of two functional concepts (function parameterization and lazy evaluation) within an object-oriented language allows combining the safety and economic usage of internal iteration with the flexibility and client control of external iteration. Sample code is provided using the statically typed Eiffel language.

1 Introduction

Collections play an important role in software design. Slightly surprisingly, the case on how to organize collection libraries and provide iteration facilities for them has not been closed yet. One reason for this are differences in the implementation languages used [22]. But also for a single language alternative designs compete with each other [30]. In the following we investigate how to design a general iteration scheme that is both flexible and easy to use. This paper compares the two main approaches to iteration, internal and external iteration, and arrives at a synthesis that virtually retains the advantages of both. The description of the problem is followed by a solution whose details and advantages are described in Sect. 4. The concluding remarks summarize and examine the implications for language design.

2 Issues in Iterator Design

Collections of data elements, such as lists, trees, graphs, etc., are without doubt very useful in the design of systems and for the implementation of algorithms. Although we often want to treat a collection as a single entity we also frequently need to individually access the contained data elements. The established mechanism for accomplishing this is an iterator, allowing us to step over all data elements until all have been visited. Whether we just want to access the elements, e.g., for printing (read iterator), search for a particular element satisfying certain criteria, modify the elements, or create a new collection of elements (write iterator), depends on the intended purpose but the same basic iteration scheme underlies all these examples.

It is clearly not an option to let clients iterate over collections using knowledge about collection internals. Each client iteration would be subject to change when collection implementations change. Evidently, an iterator abstraction is necessary that allows accessing the elements of a collection independently of their respective internal representation. With the decision for a dedicated iteration abstraction, however, the following issues must be resolved:

- *How to combine iteration and action?* How do we combine an iteration algorithm (e.g., a simple loop) with a particular function or action (e.g., print an element)? The iterator client may do it by calling both, or the iterator could be subclassed for each function, or the iterator could accept a function.
- *Who knows how to iterate a collection?* Where is the best place to put the iteration logic? Will the collection explore itself or is it better to externalize such functionality?
- *Who controls the iteration?* Who is in control of advancing the iteration and who may decide to stop prematurely, i.e., avoid a full collection exploration?
- *How to support multiple iteration strategies?* Non-linear collections, such as trees and graphs support different traversal strategies like breadth-first-search and depth-first-search with variations such as pre-order, in-order, and post-order traversal. How do we allow a choice between alternatives without excessively widening the collection's interface?
- *How to allow several iterations in parallel?* In which way can we support multiple iterators using a shared collection? Two or more clients may want to process the same collection with interleaving execution. It may even be the case that a reading iterator uses the results of a write iterator that changes the elements of a structure.

Some proposed solutions can easily be dismissed:

– Combining iterator and iteration function with inheritance [19,24,20] does not scale with respect to the number of traversal alternatives and iteration functions. Further problematic issues are described in [14].
– Equipping collections with a cursor that allows iteration of the whole collection (e.g., lists [23,29]) is problematic in the presence of multiple iterations. Even if interference is prevented by providing a cursor stack, which clients use to push and pop cursors, the resulting scheme is inelegant and error prone [12]. This suggests that the state of iteration should be kept outside the iterated collection.
– Schemes relying on co-routines or specializations thereof [18,25], are not easily applicable in languages without these mechanisms.

We are left with two fundamentally different approaches:

1. External iterators place the iteration logic outside of collections and provide clients with an interface to start, advance, and inquire the actual element of an iteration:

```
from books.start until books.exhausted -- initialize & test
loop
  io.putstring(books.item.title);      -- iteration action
  books.forth;                         -- advance iterator
end
```

2. Internal iterators are typically a part of the collection's interface. When given an iteration function they autonomously perform the traversal, thus, releasing the client to provide a control structure:

```
books.do(printTitle); -- printTitle prints argument's title
```

2.1 Internal Iteration

Internal iteration corresponds to functional mapping, i.e., the parameterization of iterators with iteration functions, and there are a number of arguments in favor of it: It adheres to the maxim "Write a Loop Once" [20], i.e., code the traversal control once inside the collection and let all clients rely on it. Hence, the duplication of virtually identical code in clients for stepping over a collection is avoided.

Traversal strategies typically rely on internal collection details for stepping over collections [25] and it is easier to use a recursive method for descending a collection than to memorize an access path externally [8].

A particular expensive incident due to external iteration was caused when the Mariner space-probe was lost due to an error in a loop [26]. In the FORTRAN code of Fig. 1 the dot should have been a comma. As it happened, just the value `1.3` was assigned to `DO3I` and no iteration took place at all. An internal iteration might not have been applicable, but at least the above example demonstrates that (fatal!) errors can be introduced in even the most simple loops.

```
DO 3 I = 1.3
```
Code to be executed with I=1, 2, 3.

Fig. 1. Code example "Goodbye Mariner"

All the above observations argue in favor of making iteration an autonomous operation of the collection, but:

> *"External iterators are more flexible than internal iterators. It's easy to compare two collections for equality with an external iterator, for example, but it's practically impossible with internal iterators. Internal iterators are especially weak in a language like* C^{++} *that does not provide anonymous functions, closures, or continuations like* SMALLTALK *and*

CLOS. *But on the other hand, internal iterators are easier to use, because they define the iteration logic for you. [8].”*

– GOF Group

Definition *Closure*

A closure is a function equipped with a map from variable names to values. Hence, a closure is a function that has partially or fully received its arguments and awaits its evaluation.

A number of other authors agree that either builtin closure support is needed to use internal iterators [2,12,8] or that internal iterators are inflexible to the extent of disallowing the comparison of two collections [25,12,8,5,27].

Also, it seems that clients know best when an iteration can be stopped, e.g., when an element has been found, and internal iterators do not account for this[1].

2.2 External Iteration

The Iterator pattern [8] implements external iteration and resolves a number of issues: The combination of iterator and iterator function is trivial, since the client calls both in a dedicated loop (see code in Sect. 2). Pattern Iterator gives the client full control of iteration advancement and termination. It, furthermore, allows multiple traversal strategies and multiple iterators on the same collection. It is weak on requiring clients to duplicate control structures, time and again. It also may force collections to provide a (possibly protected) interface to allow their efficient scrutinization for traversal. Finally, an external iterator has to keep track of the iteration state – which may involve book keeping of paths into tree-like collections.

2.3 Comparison of Internal and External Iteration

Diametrical to an external iterator an internal iterator is strong on localizing the control to one loop, information hiding of collection internals, and competence of collection exploration. Unfortunately, it requires closures for combining iterator and iteration function, takes termination control out of the client’s hands, and allows only one iteration at a time. Furthermore, multiple traversal strategies cause the collections interface to be bloated with iteration methods. Table 1 summarizes the comparison of external with internal iteration.

The last point in Table 1 refers to the fact that external iterators typically depend on the properties of the collections they traverse (e.g., trees need different iterators than linear collections). Therefore, it is common to observe a class hierarchy of iterators paralleling a collection hierarchy [8,24].

While internal iterators seem to be more faithful to software design matters than external iterators, they apparently let clients down in terms of straightforward (active) usage and flexibility.

[1] This is not true for SMALLTALK where blocks returning a value cause control to be passed back to the iteration client.

Table 1. External versus internal iteration

	Iterator kind	
Iteration framework aspect	*external*	*internal*
combination of iteration and action is trivial	✓	✓[⋆]
record keeping of iteration state is straightforward		✓
no special access interface to collection is required		✓
no explicit loop needed for client iteration		✓
client may stop iteration early	✓	✓[†]
iterating multiple collections in lock-step is easy	✓	
traversal alternatives do not bloat collection's interface	✓	
multiple iterations sharing one collection	✓	
no parallel hierarchy of iterators and collections		✓

[⋆] Support for closures required.
[†] By inelegantly passing a `continue?`-flag from function to iterator.

3 Stream Based Iteration

An elegant way to resolve the forces in Table 1 is to

– provide a way to flatten collections to a stream of data and
– iterate internally over (multiple) streams.

Definition *Stream*

A stream is a possibly infinite list. Streams allow access to the first element and to the tail of the stream only. Streams, hence, may produce elements only when they are actually required. Clients can not distinguish between streams that already know all their elements, produce them as blocks in a buffered fashion, or produce them element-wise by demand. Streams can, therefore, be regarded as lazy lists.

For instance, a tree is first tranformed into a linear stream and then this stream is consumed by an internal[2] iterator. An element of the intermediate stream might contain a collection element or a lazy exploration of further parts of the collection, i.e., an iteration continuation yielding further collection elements when inspected (see Fig. 2 and Sect. 4.3, *Traversal alternatives*). The stream consumer, hence, can decide which parts are to be explored next.

While all streams share the same interface, it depends on the collection type how many iteration continuations it produces as stream elements. The stream consumer selects a traversal alternative (e.g., pre-order, in-order, post-order) by evaluating the iteration continuations in the corresponding order.

Intriguingly, this approach manages to resolve all the forces discussed above (see Sect. 4.3), apart from two remaining crucial points:

[2] Note that "internal" now only characterizes the passive role of the client rather than the location of the iteration interface with regard to the collection.

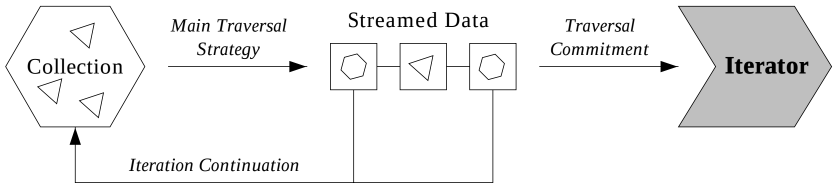

Fig. 2. Iteration topology with lazy streams

1. How to circumvent the need for closure support?
2. How to avoid the inflexibility of internal iterators?

3.1 Eliminating the need for Builtin Closures

The first problem can be immediately solved by emulating closures. In an object-oriented language it is very easy to use objects as function closures. Simply define a **Function** interface (also see Fig. 9) with an application method, e.g.,

```
deferred class Function[In, Out]   -- argument/result types
feature
  infix  "@" (a : In) : Out is deferred end; -- application
end
```

Subclasses then refine the generic type parameters and provide the implementation for various functions. For instance,

```
class Square
inherit Function[Integer,  Integer];
feature
  infix "@" (arg : Integer) : Integer is
  do
    Result := arg * arg;
  end;
end
```

If a function requires two or more arguments then the result of the first application is a further function object awaiting application to the rest of the arguments. Hence, this design nicely incorporates partial parameterization [13] (see the appendix or the full, clickable HTML source code [15] for details).

3.2 Making Internal Iteration More Flexible

The inflexibility of internal iterators, e.g., to compare two collections with simultaneous iterations, is easily fixed with a small but very effective idea. Indeed, it is practically impossible to consider a second iteration while an internal iteration focuses on its sole collection iteration. The problem, however, is easily resolved by generalizing internal iteration from one to many collections (see Fig. 3). An internal iterator, consuming two number streams, for instance, takes a function with two parameters and applies it to the two foremost numbers. Then, both streams are advanced simultaneously and the next application will be on the two following numbers.

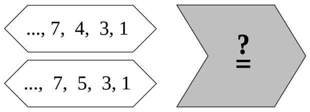

Fig. 3. Comparison of two collections in lock-step

To generalize from two to any number of input streams the processing function (e.g., equality) must process a list of arguments. Then, the arity of the function is always one (one list) but the real number of arguments is determined by the list length. Restricting all arguments to conform to the list type makes this approach statically type safe with no need for dynamic type checks (reverse assignment attempts or downcasts). All that is needed is a (genericity) mechanism that allows restricting the element type of a structure.

Transposing the list[3] of input streams yields a structure ready to be processed row by row (see Fig. 4).

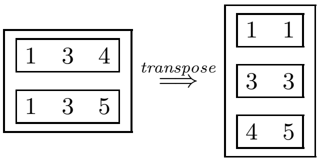

Fig. 4. Transposing lists

In case the input streams have different length then the shortest one determines the end of iteration. See the appendix for the Eiffel implementation of transpose.

Note that transposition, i.e., the processing of multiple input streams in lock-step is just one possible iteration scheme. A different operator might consume the input streams with individual speeds, for instance, to implement merge sort.

The next step towards a definition of a multi-collection iterator is to define a function that will process the rows of the transposition result. As it applies (maps) a function to each row in the transposed structure we call it transmap.

[3] We say “list” when referring to the ordered collection of input streams for clarity of presentation only. Technically, this list will be a stream too.

Definition *Fold*

A fold function processes a list with a function. Folding a list can yield any result type including a transformed version of the input list. Typically, however, a fold reduces a list to a single result by applying the function to the first element and the result of folding the rest of the list. In order to define the value of folding an empty list, an initial value is passed as an argument to fold. A transformed result list may be obtained by using a function that transforms input elements individually and builds up a list from the results.

The function argument to transmap will typically reduce (fold) one row into a single result (see the appendix for an implementation of fold). For instance, a product calculation is possible by reducing the list with multiplication. It is now straightforward to calculate the element-wise products of two integer lists (see Fig. 5).

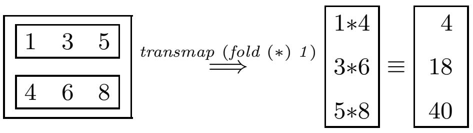

Fig. 5. Computing a list of products

Until now, function transmap allows varying the function to combine the elements of multiple collections. However, we also need to iterate over the results. For instance, it is only a matter of summing up the values in the above result list to obtain the inner product of the two argument lists. Or, returning to our original example, when comparing two collections the results of comparing corresponding elements must be reduced to a single result (see Fig. 6).

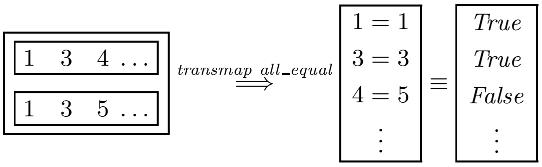

Fig. 6. Transmapping equality

Obviously, we need to reduce the result list with the logical "*and*" operator to obtain a single equality result. Likewise, the result list of Fig. 5 requires reduction with "+" to obtain the final inner product. Therefore, as the final step to define the multi-collection iterator we reduce the results of processing the rows to a single result. Here is how transfold[4] can be expressed with EIFFEL assuming `streams` to be the list of input streams (see the appendix for its full implementation):

```
Result := fold @ foldFunc @ init @
          ((map @ mapFunc) @ transpose(streams));
```

For instance, the application of transfold to "+" and (fold $*$ 1) on the argument streams of Fig. 5 will yield the result 62. Expressed with EIFFEL code:

```
Result := transfold @ plus @ 0 @ (fold @ times @ 1) @ streams;
```

Functions `foldFunc`, `mapFunc`, and value `init` correspond to functions f, g, and value a of Table 2 respectively. It lists all transfold parameters with their type and meaning.

Table 2. Transfold's arguments

Parameter	*type*	*purpose*
g	$[a] \rightarrow b$	the function that is applied (mapped) to each row of the transposed argument, transforming a row to an element of the intermediate result of type b.
a	c	the initial element for producing the final result, used as the induction base for an empty list.
f	$b \rightarrow c \rightarrow c$	function that finally reduces the intermediate result of element type b to a result of type c, using the initial element.
	$[[a]] \rightarrow c$	resulting type of transfold after all arguments but the last are supplied. Transforms a matrix (list of streams) with element type a into a result of type c.

Given a function *all_equal* that checks whether all elements in its argument stream are equal, the application of transfold to *and* and *all_equal* on the argument streams of Fig. 6 will yield *False*.

Note that, for example, in the context of comparing collections, lazy transposition and reduction functions allow stopping the exploration of the (possibly infinite) argument streams when a non-equal argument pair has been found.

To illustrate the possible type changes from argument to result let us calculate the sum of all row products from a matrix[5] (see Fig. 7), using (very academically) three different number types. We use

```
innerProd := transfold @ plus @ 0.00 @ (fold @ times @ 1.0);
```

[4] Transpose and fold.

[5] A matrix shall be represented as a list of row streams.

where the input matrix contains integer elements, 1.0 denotes a real, and 0.00 denotes a double. Hence we establish the mapping [$a \mapsto$ *integer*, $b \mapsto$ *real*, $c \mapsto$ *double*]. See Fig. 7 for the calculation process.

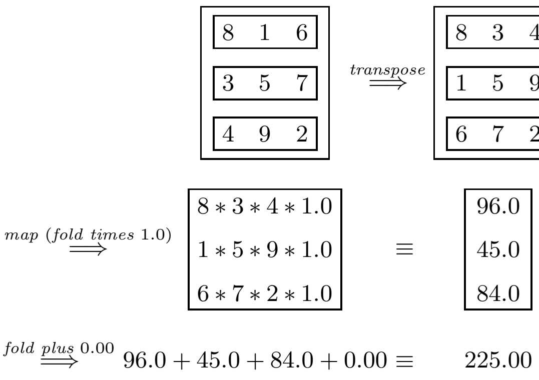

Fig. 7. Transfolding the inner product of a matrix

Folding does not have to imply reduction, though. Using functions *reverse* and *add_back* that establish the mapping [$a \mapsto$ *integer*, $b \mapsto$ [*integer*], $c \mapsto$ [[*integer*]]], we may transpose a matrix along its minor axis (see Fig. 8).

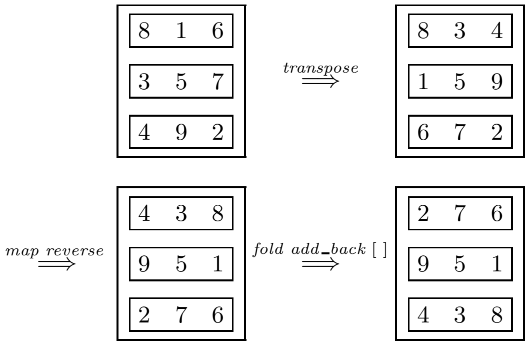

Fig. 8. Minor axis matrix transposition

We used a mixture of illustrations, functions, and sketches of EIFFEL code to demonstrate transfold and its associated functions as clearly and concisely as possible. With the help of two functional patterns (Function Object and Lazy Object [16]), it is very easy to fully implement the solution in an object-oriented language (see also the clickable HTML code [15]).

4 Stream Based Iteration Framework

The following sections describe the proposed iteration framework with a class diagram, a list of participants with their responsibilities, and the sequence of events for an iteration. Section 4.3 concludes the description with a list of framework properties.

4.1 Framework Participants

Figure 9 shows all participants in the iteration framework and their relationships. Dashed boxes at the top right hand corner of a box indicate generic classes.

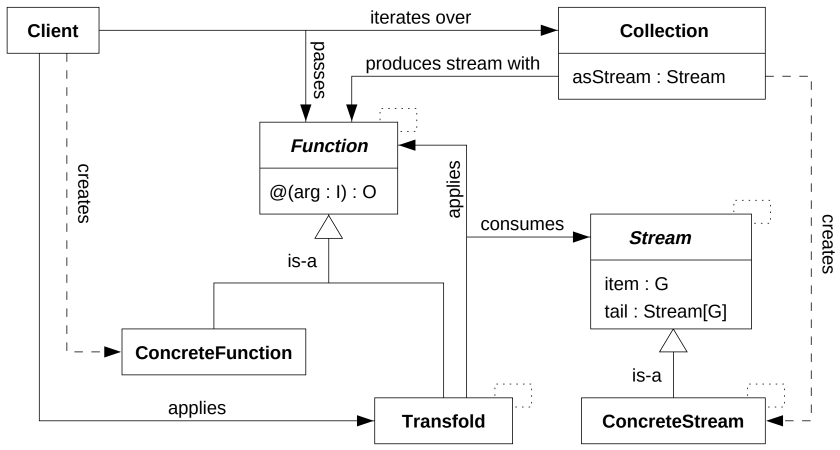

Fig. 9. Structure diagram

- **Client**
  - requests a **Collection** to provide a **Stream** of itself.
  - passes two **Function**s and a value as parameters to **Transfold**.
- **Collection**
  - provides a **ConcreteStream**, containing a flattened version of itself.
  - uses a lazy **Function** to produce a **ConcreteStream**.
- **Function**
  - provides an application interface for all functions including the stream building function, the **Transfold** parameters, and **Transfold**.
- **Stream**
  - provides an interface to access any concrete streams.
  - implements a lazy, infinite list semantics.
- **Transfold**
  - takes two **Function**s and a value as processing parameters.
  - transforms its input (a **Stream** of **Stream**s) to an arbitrary result type.

4.2 Sequence of Events

- A client requests a collection to flatten itself to a stream.
- The collection's asStream method and a lazy function mutually call each other to explore the collection lazily, while producing a stream.
- The client uses or creates two function objects, which it passes – along with an initial value – to a transfold object.
- The transfold object lazily accesses the collection stream by using the stream interface, e.g., operations "item" and "tail". While the stream is accessed, the argument functions to transfold are applied accordingly.

The purpose of the above details is to present explicitely the mechanics of the iteration framework. Real clients, however, should not have the burden of asking the collection for a stream and then of feeding it into an iterator. It is more reasonable to use a method in an abstract collection interface that takes all transfold parameters and does the stream creation and feeding behind the scenes.

4.3 Framework Properties

► *Abstraction.* Accessing the elements of a collection does not expose its internal representation.
► *Locality.* A particular operation can be performed by just passing function objects, without requiring inheritance or client control structures.
Since there is only one iteration loop, used by each client, loop-related errors are much easier to avoid and to discover. More time can be spent on the validation of a single loop[6] and any errors are removed for all clients.
► *Multiple Traversals.* When a collection is to be iterated by multiple clients in alternation it is possible to share the (read-only) collection stream for independent consumption by multiple clients. Hence, any exploration effort by the collection is beneficial to all consumers. Once explored, a subpart does not need to be traversed again due to the call-by-need semantics of streams. Whenever a read iterator needs to see the results of a write iterator we propose to use a chain of iterators where collections are not mutated but intermediate results are produced (see Fig. 10). Instead of destructively changing the contents of one collection – which can cause considerable trouble in the presence of sharing – an intermediate collection which contains the new data is produced to be subsequently consumed by the read iterator.
► *Connectivity.* Stream producing transfolds allow cascaded transformations[7] and converting one collection into another one, possibly with intermediate processing, using collection constructors with stream arguments to build collections (see Fig. 10).
Intermediate results never exist in their entirety due to lazy evaluation. The demand driven characteristics of lazy evaluation only ever creates as much of

[6] By referring to "loop" we also include the stream generation processes.
[7] Note that one transfold can perfectly just consume a single stream.

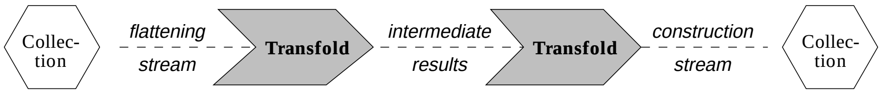

Fig. 10. Transformation chain

the intermediate results as necessary to pass the desired information through. An intuitive picture of the process is to imagine the source collection to be a wool bale being unwound, while just the thread is passed through the transfolds, to wind the result wool bale. Maintaining the internal shape of a collection during a transformation as shown above is easier with destructive updates. However, the collection constructor of the destination collection can use the order of the arriving elements and its own structure invariants to create a collection that is behaviorally equivalent to the original.

► *Termination control.* Both transfold and the client are in control of iteration advancement and termination. Through the use of lazy stream processing functions, the collection exploration is completely demand driven. When a stream processing function does not evaluate its second argument – e.g., an *And* does not need to examine the second argument, if the first is already *False* – the whole transfold process stops. This scheme is far more elegant than letting an iteration function return a `continue?`-flag, as designed in the internal version of the Iterator-pattern [8].

► *Traversal Alternatives.* Collections produce a stream containing elements and iteration continuations which are again represented by streams (see Fig. 2). For instance, graphs can produce a stream representing a forest (a stream of tree streams). Stream consumers decide in which order they consume elements and explore continuations. Hence, it is easy to support a variety of traversal strategies outside of collections (see Fig. 11).

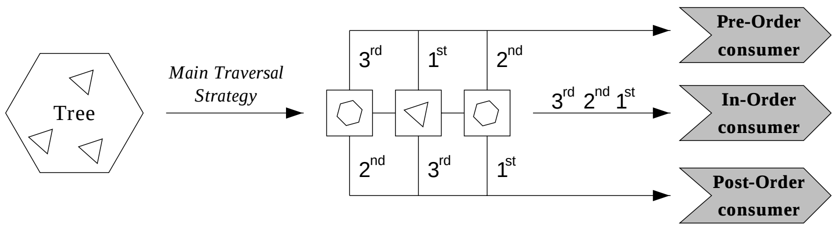

Fig. 11. Deriving traversal alternatives

One can even dynamically dispatch on traversal strategies. Through lazy exploration of the collection the stream processing functions (passed as arguments to a transfold object) are in command of the exploration order.

Not all traversal alternatives, however, can be derived from just one main traversal strategy while retaining lazy exploration. Assume, for instance, a linear structure providing the elements one by one. It is then not possible to derive a backward iteration from this main traversal strategy without fully exploring the structure to arrive at the last element. In such cases it is possible to provide alternative asStream methods in the collection interface.

► *Separation.* The best of both external and internal iteration are combined by separating the *exploration* of a collection and the subsequent *consumption* of the exploration result.

+ Since iteration (consumption) is defined outside collections, their interfaces can be kept small. The only trace of an iteration is an asStream method, which is of general interest anyway (see bullet "*Streamable Collections*" below).
+ The responsibility to explore a collection is assigned to the most competent instance, the collection itself. The collection may use all its internal knowledge and recursive calls – thereby memorizing an exploration stack – to perform its exploration. Contrast this with a much more complicated scheme implementing recursion with an explicit stack and counting (observing) node removals [38].

  It is possible to separate consumption and exploration without having the overhead of a full exploration, because the intermediate stream is lazily produced. For a general account on pattern Lazy Object see [16].
+ Streams work as a *lingua franca*[8] between collections and iterators. Both stream consumption (iteration) and generation are easy to vary. Iteration schemes, such as transfold, must be defined only once, for all streamable collections.

  Special traversal orders may depend on stream organization but not on collections, which is a useful indirection to decrease coupling.

► *Streamable Collections.* The asStream method of data collections can also be used for many other purposes, such as a persistence or net-transfer protocol mechanisms.

Collections may be transfered into each other by means of an intermediate stream. For instance, streaming a **Bag** to a **Set** is an elegant way to remove duplicates. No special mechanisms, e.g., the Serializer pattern [31], will be needed anymore to support persistence in this manner.

This also implies that there is a uniform way to construct collections, for example, from constants. Any collection type that allows manifest constants in the syntax of a language (e.g., arrays), could be used to be transformed to the desired collection type.

► *Versatility.* Folding may implement a wealth of operations, for instance for lists: sum, product, average, max, min, map, filter, reduce, reverse, append, exists, all, variance, horner, etc. According to [37], 60% of the code in the Fortran Scientific Subroutine Package fits neatly into the maps, filters, and accumulations (i.e., transfold) paradigm.

[8] Agreed language of communication.

► *High-level Mindset.* A capable, high-level operation like transfold enables approaching problems with a much more powerful decomposition strategy compared with a procedural paradigm, restricted to e.g., array indexing. Timothy Budd tells an anecdote of a FORTRAN programmer who, predetermined to think in terms of loops and array access, designs a three-level nested loop to find a pattern repetition in a DNA sequence. His algorithm's complexity turns out to be $O(M * N^2)$, where M = pattern length and N = sequence length. An APL programmer, thinking in high-level operations like vector to matrix conversion, sorting, and matrix reduction (all akin to and expressible with transfold), arrives at a solution with complexity $O(M * N \lg N)$ [5]. As the anecdote suggests, high-level operations allow problems to be approached from a different, valuable perspective.

► *Choice of Style.* In cases where no predefined iteration scheme, like transfold, seems appropriate, it is possible to consume a collection stream with an external iterator, i.e., to write a control loop which consumes the stream. The essence of the proposed iteration framework lies in stream producing collections, function objects, and lazyness. The transfold operator is just one of many possible.

► *Fixed Forms.* A view emerged from the so-called *squiggolist* school [21] aims at a formalism that does not allow users the free definition of recursive functions but supplies a limited set of functional forms, such as fold, that are well controlled and amenable to program transformations. Relying on a set of functional forms (e.g., transfold) is advantageous in many ways:

– Algorithms using the forms have a concise, readily understandable structure as well as a determined complexity in time and space. Any programmer familiar with the functional forms will understand the algorithm by just looking at the essential parameterized parts.
– Well-known laws for the functional forms may be exploited to transform programs. For instance, instead of multiplying all elements of a collection by 2 and then summing them up, they can be summed up first applying the multiplication by 2 to the sum, hence only once. The general law for this transformation is a free "fold-fusion" theorem that is derivable from the signature of fold [36]. More such laws can be found in [21].
– Algorithms written as a combination of parameterized combinators can be easily varied. For instance a *tournament-sort* combinator taking two reduction strategies as parameters can express *InsertSort* [3], *TreeSort* [7], and *ParallelTournamentSort* [33], just by using different combinations of the two reductions operators [10]. *ParallelTournamentSort*, in fact, has very desirable properties, which demonstrates that using standard combinators does not necessarily imply inefficient solutions and may, on the contrary, help to discover better solutions.

► *Contra-Indication.* This iteration framework should not be used in case of very tight memory and time constraints, where the overhead of an intermediate stream and emulation of lazy evaluation is not tolerable. Management of stream elements consumes time. Stream suspensions and lazy function closures represent a memory overhead. In most cases, however, system performance should not be a problem.

5 Related Work

Although C^{++} usually promotes external iteration, there is an example of an internal iterator (`foreach`) interface in the Borland C^{++} libraries [4]. Since it builds on passing function pointers, it must use an extra, unsafe `void` type for passing parameters.

The Standard Template Library (STL) [34], provides function objects, a variety of (forward, backward, etc.) iterator types including stream iterators. However, iterators are of the external type and collections directly return iterators instead of streams with continuations.

The generic collection library for JAVA (JGL) [28] has its roots in STL. It provides an alternative to the JAVA collections API [35]. In a readers poll [30] the usage of function objects within the JGL was found to be powerful and flexible while the collections API was conceived to be more lightweight and simpler to use.

The function fold originates from functional programming [3], but is also used in SCHEME [1] and available in the SMALLTALK library [17]. SMALLTALK users do not use it frequently [32], probably because they are unfamiliar with the nature of folding and its peculiar name (inject: into:). The SMALLTALK collection library even contains a with: do: method, allowing iteration over two collections in parallel, which represents a special case of transfolding. SMALLTALK also uses streams to implement concatenation of collections efficiently. The caching effect of streams eliminates the need to repeatedly generate the prefix of sequenced concatenations like `coll1 + coll2 + coll3 + coll4`.

APL [9] is well-known for its high-level operations on vectors, matrices, and structures of even higher dimension. Three of its four primitive extension operators[9], reduction ($\texttt{f}/A$), scan ($\texttt{f}\backslash A$), and innerProduct ($A\texttt{f.g}B$), can directly be expressed with transfold. The fourth, outerProduct ($A\circ.\texttt{f}B$), is expressible with a combination of transfold and map. Function map is just a (trans-)fold with argument functions that do not reduce the input stream but only apply a function to it.

An interesting competitive iteration approach is set out by the SATHER programming language. Language support for iterators – in the form of a restricted kind of coroutines – allows defining collection exploration within collections while still enabling flexible, external iteration style, consumption [25]. The open questions are which style (passing functions or coroutine-loops) is more expressive and understandable, and whether possible code optimizations by the SATHER compiler justify the requirement for an additional language construct. In effect, the ability of lazy evaluation to defer calculations and resume control to them whenever necessary is very similar to coroutines. With coroutines, however, the emphasis is on explicitely scheduling control whereas lazy evaluation causes a more declarative, demand driven style.

Kofler investigated how to make iteration over collections which are changed during iteration a safe and unambiguous operation [12]. Using transfold's scheme

[9] These extend an operation to a collection.

a so-called *iterator adjustment* scheme [12], is particular well implementable, since the collection controls its own exploration and, thus, may adjust an exploration process according to element removal or insertion. Consequently, no registering of active iterators [12] is necessary. In cases where updates should have no effect on the iteration process, it is possible to simply iterate on a copy of the collection.

6 Conclusions

Neither external iterators nor internal iterators operating on a single collection only provide a satisfactory general iteration framework. Functional techniques (made available by the functional patterns Function Object and Lazy Object [16] for closure and lazy semantics emulation respectively), however, make internal iteration feasible for standard object-oriented languages. Function objects enable behavior parameterization, whereas lazy evaluation makes the separation of collection exploration and data consumption feasible. Combined with the idea of simultaneously processing multiple collections during one internal iteration, the result called "transfold" provides the safety and economic efficiency of internal iteration while maintaining the flexibility and control of external iteration. Transfold, however, is just one of many possible operations. Other iteration schemes may process multiple collections without imposing a lock-step processing fashion. The presented stream based iteration framework even extends into an approach for creating data from manifest constants, transforming collections into each other, and data persistence.

Some users, who are unfamiliar with the map/fold style of functional programming, may find the general nature of transfold intimidating and awkward to use. Nevertheless, transfold can be used to build specialized iteration operations that are easier to use. With just one iteration primitive (folding) many operations can be expressed (see Sect. 4.3, *Versatility*). This is in contrast to, e.g., EIFFEL's plethora of iteration features (do_all, do_if, do_while, do_until, etc.) in its iteration classes [24]. Although EIFFEL supports only one language loop construct, its library approach to iteration apparently does not allow such a minimalistic solution. While not every computable function is expressible with folding [11], it is sufficiently general to rely on it as the basic iteration principle. Fortunately, stream based iteration easily allows the introduction of new iteration schemes whenever necessary.

In general, internal iteration is better suited for parallel execution since, unlike a custom external iteration, the iteration process is guaranteed to be encapsulated [2]. In particular, functional forms such as *map* or *parallel-reduce* allow parallel evaluation. In fact, there is a tradition in the area of parallel computations to employ fixed evaluation forms called algorithmic skeletons [6].

Several improvements to current object-oriented languages would aid the construction of the transfold framework. First, instead of manually defining function object classes that just forward arguments to implement partial parameterization, the language should do this automatically. The recently proposed mechanism for delayed calls in EIFFEL supports partial parameterization to some extent. While it is possible to leave parameter positions open during the creation of a delayed call, it does not seem to be possible to obtain another delayed call from a delayed call by passing a subset of the open parameters only.

Second, using EIFFEL it is not possible to achieve transparent lazy semantics for basic types, e.g., **Integer**, because access to them is implicit, i.e., there is no access method which could defer the calculation. Although, it is possible to use wrapper classes for basic result types, which demand using a value or item method to access values (see EIFFEL's **NUMERIC_REF** classes or JAVA's approach to treat basic types as objects), this solution obviously makes lazy semantics visible to clients. Language support for lazy semantics could fix this and in addition eliminate the need for an extra lazy stream generating function object. Lazy collection methods could achieve lazy exploration on their own, thus implementing collection exploration at the best possible location, inside collections only.

Third, although standard parametric polymorphism is sufficient to support transfold with regard to its flexible types, type system support could be much better. For instance, the EIFFEL implementation forces us to include the intermediate (row result) type in the list of generic parameters for **Transfold**. This is unavoidable because it is the only way to ensure that the result type of the row processing function (g) and the input type of the row results reduction function (f) match. EIFFEL does not provide access to the actual value of the generic type parameter of one function in order to match it with the argument type of another. Hence, the introduction of dependent types would further support the definition of transfold in the presence of static typing. Interestingly, the way C^{++} treats generic type parameters implicitly, does not cause the above complication.

While the usefulness of function objects in object-oriented languages is generally acknowledged by now, the above observations also strengthen the case for integrating a seemingly exotic functional concept, i.e., lazy evaluation, into object-oriented languages. The benefits of using a complete functional pattern system and its implications on language design are described in [16].

Acknowledgments

The author would like to thank Thilo Kielmann for his feedback, John Hopkinson for proof reading, and the anonymous reviewers for their extremely knowledgeable and helpful comments.

References

1. Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs.* The MIT Press, Cambridge, MA, London, 6th edition, 1987.
2. Henry G. Baker. Iterators: Signs of weakness in object-oriented languages. *ACM OOPS Messenger*, 4(3):18–25, July 1993.

3. Richard Bird and Philip Wadler. *Introduction to Functional Programming.* C.A.R. Hoare Series. Prentice Hall International, 1988.
4. Borland. *Borland C/C^{++} 4.0 Reference Manual.* Borland, Inc., 1994.
5. Timothy Budd. *Multiparadigm Programming in Leda.* Addison-Wesley, 1995.
6. John Darlington, Yi-ke Guo, Hing Wing To, and Jin Yang. Parallel skeletons for structured composition. In *PPoPP '95*, pages 19–28, St. Barbara, CA, July 1995.
7. R. W. Floyd. Treesort (algorithm 113). *CACM*, December 1964.
8. E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides. *Design Patterns: Elements of Object-Oriented Software Architecture.* Addison-Wesley, 1994.
9. Edward Harms and Michael P. Zabinski. *Introduction to APL and computer programming.* Wiley, 1977.
10. Aaron Kershenbaum, David Musser, and Alexander Stepanov. Higher-order imperative programming. Technical Report 88–10, Rensselaer Polytechnic Institute Computer Science Department, April 1988.
11. Richard B. Kieburtz and Jeffrey Lewis. Programming with algebras. In *Advanced Functional Programming*, number 925 in Lecture Notes in Computer Science, pages 267–307. Springer, 1995.
12. Thomas Kofler. Robust iterators for ET++. *Structured Programming*, 14(2):62–85, 1993.
13. T. Kühne. The function object pattern. *C++ Report*, 9(9):32–42, October 1997.
14. Thomas Kühne. Parameterization versus inheritance. In Christine Mingins and Bertrand Meyer, editors, *Technology of Object-Oriented Languages and Systems: TOOLS 15*, pages 235–245, Prentice Hall International, London, 1995.
15. Thomas Kühne. Transfold Eiffel code implementation: Classes & HTML, `http://www.soc.staffs.ac.uk/~cmttk/transfold.html`, November 1998.
16. Thomas Kühne. *A Functional Pattern System for Object-Oriented Design.* ISBN 3-86064-770-9, Kovač Verlag, Hamburg, 1999.
17. Wilf LaLonde. *Discovering Smalltalk.* Benjamin / Cummings Publishing, 1994.
18. Barbara Liskov and John Guttag. *Abstraction and Specification in Programm Development.* MIT Press, 1986.
19. O. L. Madsen, K. Nygaard, and B. Möller-Pedersen. *Object-Oriented Programming in the BETA Programming Language.* Addison-Wesley and ACM Press, 1993.
20. R. Martin. Discovering patterns in existing applications. In J. O. Coplien and D. C. Schmidt, editors, *Pattern Languages of Program Design*, pages 365–393. Addison-Wesley, 1994.
21. E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Proceedings of the 5th ACM Conference on Functional Programming Languages and Coomputer Architecture, Cambridge, Massachusetts*, LNCS 523, pages 124–144. Springer Verlag, August 1991.
22. Gisela Menger, James Leslie Keedy, Mark Evered, and Axel Schmolitzky. Collection types and implementations in object-oriented software libraries. In *The* 26th *TOOLS conference USA '98*, 1998.
23. Bertrand Meyer. *Object-Oriented Software Construction.* Prentice Hall, Englewood Cliffs, NJ, 1988.
24. Bertrand Meyer. *Reusable Software.* Prentice Hall, 1994.
25. Stephan Murer, Stephen Omohundro, and Clemens Szypersky. Sather Iters: Object-oriented iteration abstraction. Technical Report TR-93-045, ICSI, Berkeley, August 1993.
26. P.G. Neumann. Risks to the public in computer systems. *ACM Software Engineering Notes 11*, pages 3–28, 1986.

27. Peter Norvig. Design patterns in dynamic programming. Presentation at Object World '96, May 1996.
28. ObjectSpace. JGL – the generic collection library for java v3.0, http://www.objectspace.com/products/jgl/, 1997.
29. Stephen Omohundro and Chu-Cheow Lim. The sather language and libraries. Technical Report TR-92-017, ICSI, Berkeley, March 1992.
30. Reader poll. Schould Sun scrap its collections API in favor of the more robust JGL?, http://www.javaworld.com. *Java World*, August 1998.
31. D. Riehle, W. Siberski, D. Bäumer, D. Megert, and H. Züllighoven. Serializer. In R. C. Martin, D. Riehle, and F. Buschmann, editors, *Pattern Languages of Program Design 3*, Reading, Massachusetts, 1997. Addison-Wesley.
32. David N. Smith. *Dave's Smalltalk FAQ*. dnsmith@watson.ibm.com, July 1995.
33. A. Stepanov and A. Kershenbaum. Using tournament trees to sort. Technical Report 86–13, Polytechnic University, 1986.
34. A. Stepanov and M. Lee. The standard template library. ISO Programming Language C++ Project. Doc. No. X3J16/94-0095, WG21/NO482, May 1994.
35. SUN. Java collections API, http://java.sun.com/products/jdk/1.2/docs/guide/collections/overview.html, 1997.
36. Philip Wadler. Theorems for free! In 4^{th} *International Symposium on Functional Programming Languages and Computer Architecture*, London, September 1989.
37. Richard C. Waters. A method for analyzing loop programs. *IEEE Transactions on Software Engineering*, 5(3):237–247, January 1979.
38. M. A. Weiss. *Algorithms, Data Structures and Problem Solving with C++*. Addison Wesley, 1996.

Appendix

The following code sketches should provide enough information concerning the implementation of transfold with Eiffel. A complete set of all required classes and a working test class, however, can be obtained from the author's homepages [15].

Section 3.2 discussed the calculation of the inner product of a matrix (see Fig. 7). We can code the corresponding inner product operation in Eiffel as:

```
local
  ip : Function [Stream [Stream [Integer]], Double]
...
  ip:=transfold @ plus @ 0.00 @ (fold @ times @ 1.0);
```

We use the same type progression (integer, real, double) for the intermediate results as in Sect. 7. Therefore, function objects `times` and `plus` must promote from integer to real and real to double respectively.

The `ip` function can be now be applied to a matrix. So, constructing integer streams with `addItem` and building the list of input streams with `addStream` –

```
vec1, vec2, vec3 : Stream [Integer];
vecs             : Stream [Stream [Integer]];
...
```

```
vec1:=addItem @ 8 @ (addItem @ 1 @ fromConst (6));
vec2:=addItem @ 3 @ (addItem @ 5 @ fromConst (7));
vec3:=addItem @ 4 @ (addItem @ 9 @ fromConst (2));
vecs:=addStream @ vec1 @
      (addStream @ vec2 @ (addStream @ vec3 @ void));
```

– the following function applications yield the results given in Fig. 12.

```
io.putdouble (ip @ vecs);
io.putdouble (ip @ vecs.tail);
```

$$225 = \texttt{ip} @ \texttt{vecs} \left\{ \begin{array}{l} 8\ 1\ 6 \\ \left.\begin{array}{l}3\ 5\ 7\\ 4\ 9\ 2\end{array}\right\} \texttt{ip} @ \texttt{vecs.tail} = 71 \end{array}\right.$$

Fig. 12. Inner product application

The row processing function **Fold**, used in the implementation of `ip` and **Transfold** is implemented (omitting the preceding argument collection classes) as:

```
class Fold2[B, C]                      -- B = stream element type
inherit Function[Stream[B], C];        -- C = fold result type
creation make
feature
  func : Function[B, Function[C,C]]; -- function argument
  init : C;                          -- initial value argument

  make(i : like init; f : like func) is -- called by Fold1,
  do                         -- whose application method "@"
    init:=i;                 -- receives "f". Fold1's make is
    func:=f;                 -- is called by Fold's application
  end;                       -- method, which received "i".


  infix "@" (stream : Stream[B]) : C is
  local fold : expanded Fold[B, C];
  do
    if stream=void then           -- end of input stream?
      Result:=init;               -- yes, return initial value.
    else
      Result:=func @ stream.item @   -- no, apply recursively.
              (fold @ func @ init @ stream.tail);
    end;
  end;
end
```

With the help of **Fold** and **Map**, implementing the body of **Transfold** is straightforward:

```
class   TransFold3[A, B, C]           -- A = element type
inherit Function[Stream[Stream[A]], C] -- B = reduction result
        StreamUtility[A]              -- C = transfold result
creation make
feature
  foldFunc : Function[B, Function[C, C]]; -- result processing
  init     : C;                           -- initial value
  mapFunc  : Function[Stream[A], B];      -- row processing

  make(f : like foldFunc; i : like init; m : like mapFunc) is
  do
    foldFunc:=f;
    init:=i;
    mapFunc:=m;
  end;

  infix "@" (streams : Stream[Stream[A]]) : C is
  local
    map  : expanded Map[Stream[A], B];
    fold : expanded Fold[B, C];
  do
    Result:=fold @ foldFunc @ init @
            ((map @ mapFunc) @ transpose(streams));
  end; ...
```

The use of `expanded` is a language idiom to save the otherwise necessary explicit attachment of an object to `map` and `fold`.

The `transpose` method of class **TransFold3**, nicely demonstrates a high-level, functional, internal iteration style, which is possible using streams and functional forms such as map and fold.

```
transpose(rows : Stream[Stream[A]]) : like rows is ...
do
  newRow:=mapToHeads @ head @ rows;   -- collect all row heads
  tails:=mapToTails @ tail @ rows;    -- collect all row tails

  if (fold @ oneEmpty @ False @ tails) then -- row exhausted?
    tails:=void;                    -- yes, end of result rows
  else
    tails:=transpose(tails);        -- no, transpose the rest
  end;

  Result:=addStream @ newRow @ (tails); -- build result
end
```