Solid: Library Overview

Greg Rosenbaum

April 20, 2013

Library Version: 0.3 (alpha)

Contents

1 Prelude & On Immutability **2**
- 1.1 Implicit copy . . . 2
- 1.2 Total control over internal state . . . 3
- 1.3 Implicit thread-safety & atomicity . . . 3
- 1.4 Implicit, efficient history . . . 3
- 1.5 Using the value abstraction for collections . . . 3

2 The Collections **4**
- 2.1 Flexible List . . . 4
- 2.2 Vector . . . 4

3 Library Features **5**
- 3.1 Implicit optimization . . . 5
- 3.2 Permissive license . . . 5
- 3.3 Compatibility features . . . 6
- 3.4 Negative indexing . . . 6
- 3.5 Fast functional operations . . . 6
- 3.6 Zero dependencies . . . 7
- 3.7 Full documentation . . . 7
- 3.8 Support for `F#` and `C#` . . . 7
- 3.9 Exceptions . . . 8
- 3.10 Naming convention . . . 8

4 C# Interface **9**
- 4.1 Easy creation of collections . . . 9

5 F# Interface **10**
5.1 Module interface 10
5.2 F# Overloads 10
5.3 Operators 11
5.4 Collection comprehensions 11
5.5 Active patterns 12

6 How Specific Operations are Handled **12**
6.1 Addition to the beginning 12
6.2 Insertion 12
6.3 Removal 13
6.4 Empty collections 13

7 Not Done Yet **13**

1 Prelude & On Immutability

`Solid` is a library of fully persistent and immutable collections.

This means that once a `Solid` collection has been created, it cannot be modified. This is not a facade or a limited interface; the collections expose no operation and have no algorithms for modifying them in-place.

They are best compared to strings. Although strings are immutable, they nevertheless implement operations such as `Substring` and `Replace`. These return a new string, with the specified modifications. The fact strings are immutable is very handy.

Strings however employ direct memory copies in the implementations of these methods. This is efficient for small strings, but terribly inefficient when dealing with large ones. `Solid` collections employ *structural sharing*, where bits of the old collection are shared with the new one.

Here are some reasons for using immutable, persistent collections in general. If you're not interested, skip to Section 2

1.1 Implicit copy

Immutable and persistent collection can be copied implicitly, as an instantaneous $O(1)$ operation. This is because you only need to copy the reference to the collection (a simple assignment). This is similar to the way you copy a number; you simply assign a new variable to its value.

Besides this, many operations which normally have to involve copying data, such as concatenation, splitting, subsequences, insertion, and so forth, can perform a lot better for persistent collections because they already have complex and efficient structural sharing algorithms.

1.2 Total control over internal state

An object that stores its internal state in an immutable collection has complete control over it, because it has complete control of all instance fields defined in its body.

When a `readonly` field is initialized with an immutable object, it *cannot* be modified in any way. This is also similar to what happens when you make a numeric field read-only.

When a property exposes a `get` accessor but not a `set` accessor for an immutable collection, this assures that the collection will never be changed by the caller.

Not only do you gain these advantages, but the caller can nevertheless access the full functionality of the collection. It functions as well when it's readonly as when it's not.

1.3 Implicit thread-safety & atomicity

Operations executed on an immutable collection are implicitly thread-safe. They are also atomic for most purposes.

This is because the only externally visible changes caused by the operation are a function return. All mutation occurs in the context of a method, and every thread by definition accesses different locals.

Besides that, because a reference to an object is guaranteed to be atomic (and unlike, say, `Int64`, which is not), there is no way in which the operation can be partially evaluated. Either a reference to the new collection is returned, it raises an exception, or it runs forever.

Synchronizing access to an immutable collection is also simple due to the atomic nature of an object reference, and can be controlled directly by the instance.

1.4 Implicit, efficient history

In most cases, previous versions of persistent collections aren't explicitly required and as such are likely to go to garbage collection. However, in some cases, such as to support undo or recovery operations, to maintain an accessible history, and so forth, these past versions are required.

In such a case, fully persistent and immutable data structures provide excellent space and time efficiency as compared to alternatives, and make such things easier to implement and control.

1.5 Using the value abstraction for collections

One of my big motivations in having immutable, persistent data structures is that they can be treated in many ways as values. This is very similar to strings,

which have the same quality.

The reason this is convenient is that collections *are* values. When you compare collections, you always want to know whether their values are equal. You don't normally care about the fact they have a unique identity. Their only meaning lies in whether they are empty or not, how many elements they contain, etc.

An operation on a collection is conceptually similar to an operation on a number, or a string. Here is a comparison

```
var list = Enumerable.Range(0, 1000).ToFlexList();
var num = 5; str = "Hello";
num = num + 5 + 5;
str = str + ", my name is Greg. " + "How are you?";
list = list.AddLast(1001).AddLast(1002);
```

In contrast, an operation on a mutable collection such as `List<T>` is performed on an object with intrinsic identity.

2 The Collections

This section surveys the unique features of the collections currently available in the library. These are in addition to everything discussed in Section 3.

Complexity information was deliberately simplified here.

2.1 Flexible List

`FlexibleList` is a general purpose sequential data structure that offers every operation you are likely to need.

All of these operations are very fast and due to their $O(\log n)$ complexity, they scale very slowly with the size of the collection.

This data structure is implemented as a 2-3 finger tree.

Queue operations Supports instantaneous addition, removal, and peek at either end. $O(1)$

Random access Slightly slower lookup, update, insert, and remove at any index. $O(\log n)$

Large operations Slightly slower concatenation, insertion of lists at index, slices from index to index. $O(\log n)$

This is in addition all the operations offered by all `Solid` data structures, of course.

2.2 Vector

`Vector<T>` is a more specialized collection that boasts fewer features but much better performance.

The complexity of the operations provided by this data structure seems worse than `FlexibleList<T>`, but this is only theoretical. In practice, it performs many operations faster, sometimes dramatically so.

It is implemented as an array mapped trie, utilizing arrays and efficient memory copies under the hood.

List operations Addition and removal from one end. $O(\log n)$

Random access Lookups and updates at index. $O(\log n)$.

Bulk operations Addition and removal of many elements from the end. Addition is $O(m)$ (as opposed to $n \log n$). Removal is $O(\log n)$.

Subsequence Offers a `Take(n)` operation. Doesn't support slices though. $O(\log n)$.

3 Library Features

These are the general features shared by all collections in the library, and are unlikely to change in future revisions. By the way, this section is written in `C#`. You might need to glance at Section 4.1 because it has some information on the interface.

3.1 Implicit optimization

All `Solid` collections use implicit optimization. This means that the collection will always try to optimize things behind the scenes.

For example, the `Vector<T>` offers a single `AddLastRange` method that adds many items to the end. The method takes any `IEnumerable<T>` object, but internally, it performs type tests to check whether it implements some collection interfaces that could considerably improve performance.

The `Vector` class also offers the optimized `CopyFrom` method, which loads data from an array. An array is the optimal type to load data from, so it skips much of the type checking and gets down to business faster. It's not expected to be much faster than passing an array to `AddLastRange`, though.

```
var range = Enumerable.Range(0, 1000000);
var arr = range.ToArray();
var col = range.ToList();
var vector1 = arr.ToVector();
var vector2 = col.ToVector();
var vector3 = range.ToVector();
//Same extension method, but different implementations.
```

3.2 Permissive license

This library is released under the MIT License

3.3 Compatibility features

All `Solid` collections implement `IList<T>` for compatibility with existing code. They are also aware of other collections, as explained in Section 3.1.

Note however that as immutable collections they return `true` for the property `IsReadOnly` and throw exceptions when attempting to access methods such as `Add` and `Remove`

```
IList<T> coreList = (FlexibleList<T>) new[] {1, 2, 3, 4};
var count = coreList.Count;
Vector<T> vector = new[] {0, 1, 2, 3, 4};
var list = Enumerable.Range(0, 100000).ToList();
//An internal dynamic type test optimizes this operation:
vector = vector.AddLastRange(list);
```

3.4 Negative indexing

One of my favorite features in other language is negative indexing, where a negative index is treated as position from the end. This feature streamlines many common operations, as is demonstrated by the following comparison:

```
var mutableList = new List<int> {0, 1, 2, 3, 4};
FlexibleList<int> xlist = new[] {0, 1, 2, 3, 4};
var last1 = mutableList[mutableList.Length - 1];
var last2 = xlist[-1];
```

All index parameters can take negative indexes in this way. For example, the `Slice` method of `FlexibleList<T>`, which returns a slice of the collection from index to index:

```
FlexibleList<int> xlist = new[] {0, 1, 2, 3, 4, 5, 6};
var slice1 = xlist.Slice(0, xlist.Count - 2);
//is identical to:
var slice2 = xlist.Slice(0, -2);
//We could also get a slice consisting of the last 3 elements:
var slice3 = xlist.Slice(-3, -1);
//Using positive indexes, this is a bother to write:
var slice4 = xlist.Slice(xlist.Count - 3, xlist.Count - 1);
```

As a side note, `FlexibleList<T>` also supports an indexer that takes two indexes. It works in exactly the same way as the `Slice` method.

```
var slice = xlist[0, -2];
```

3.5 Fast functional operations

All `Solid` collections have specialized implementations of many functional operations, such as an aggregation operation (called `Fold` here), a `Select` operation,

and a `Where` operation.

These specialized operations are both faster and also generate a collection of the same type as the one you started with. This feature is not currently integrated with the collection builders discussed in Section 4.1 and these specialized operations don't use lazy evaluation.

I'm hoping to change this, though, when I figure out how to retain the best of both worlds.

```
FlexibleList<T> list = new[] {1, 2, 3, 4, 5, 6};
list = from item in list
       where predicate(item)
       select item;
var result = list.Fold((fold, cur) => cur * fold);
```

There are also dedicated method such as `ForEach`, for iterating over the collection.

3.6 Zero dependencies

I've made a conscious effort to scale back dependencies and framework versions as far as possible. The library has no dependencies other than `.NET` Framework 3.5 Client Profile. The `F#` extension library also requires the `F#` core library.

I doubt I'll introduce any external dependencies, but I may increase the required framework version if I see any features that are really necessary.

3.7 Full documentation

All collections are fully documented, and there is a reference API compiled using Sandcastle. Here is a list of everything documented in the library.

1. All user-visible classes and method signatures are documented.
2. Most thrown exceptions are also documented.
3. Most members are annotated with performance information.
4. A `remarks` section featuring additional performance information and other guidelines. Implicit optimizations are usually listed here.

3.8 Support for `F#` and `C#`

I've made quite a bit of effort to make sure the library is compatible with both `F#` and `C#`, and included language-specific features for each of them. More information is available in Section 4 and Section 5. Here is a brief example:

```
//C#:
var test = from num in Enumerable.Range(0, 1000).BuildVector()
           where predicate(num)
           select 2 * num;
```

```
//F#:
let test =
    vector {for i in {0 .. 1000} do if predicate i then yield 2 * i }
```

3.9 Exceptions

`Solid` collections throw all the exceptions they should, and none of those that shouldn't.

Or at least, I've made a conscious effort to perform all error checking on the visible interface level to make sure no operation raises an exception in the invisible implementation layer, where the exception would be confusing.

This aspect of the library will improve with time, as the collections are tested.

1. `ArgumentNullException` thrown when an unexpected argument is null.
2. `ArgumentOutOfRangeException` thrown when there is a problem with one or more index arguments.
3. `InvalidOperationException` is thrown if an illegal operation was executed on an empty data structure. Also thrown in some other special circumstances.
4. `OutOfMemoryException` is not thrown by the collections (that would be silly), but it is documented in methods that could possibly throw it.

The collections do not currently define any new exception types.

3.10 Naming convention

Naming conventions in `Solid` are deliberately highly standardized. You should be able to understand what a method does by the words that compose its name. The following is a table that will, I hope, allow you to know the function of any member at a glance.

This is also true, in general, for `F#` members.

Particle	Meaning	Example
`Add`	Adds an item to the collection	`AddLast`
`Drop`	Removes an item from the collection	`DropLast`
`First`	The beginning of the collection	`AddFirst`
`Last`	The end of the collection	`AddLast`
`Range`	Refers to a sequence of items	`AddLastRange`
`Count`	A number of items	`Count`
`List`	Refers to a sequential colletion	`AddLastList`
`While`	Does something while a predicate is true	`TakeWhile`
`Slice`	A subsequence by 2 indexes.	`Slice`
`Take`	A starting subsequence	`TakeWhile`
`Skip`	Bypass a starting subsequence	`SkipWhile`

`x[i]`	Lookup an item	`this[int]`
`x[i,j]`	A subsequence by 2 indexes	`this[int,int]`
`Insert`	Inserts one item at index	`Insert`
`Remove`	Removes one item at index	`Remove`
`Where`	Filters based on a predicate	`Where`
`Select`	Projects a collection	`Select`
`Many`	More than one item	`SelectMany`
`Set`	Update by index	`Set`
`ForEach`	Iterates from first to last	`ForEach`
`Back`	Does something from last to first	`ForEachBack`
`Fold`	Similar to `Aggregate`	`FoldBack`

4 C# Interface

4.1 Easy creation of collections

`C#` doesn't support list comprehensions. It does support collection initializers though, and arrays generally have the shortest one. Unfortunately, there isn't a way to make collection initializers work for immutable data structures. The interface simply doesn't support this.

However, I think the following is a neat workaround that has its own advantages. You can imitate a list comprehension using an implicit conversion from an array.

```
FlexibleList<int> list = new[] {1, 2, 3, 4, 5};
Vector<int> vector = new[] {1, 2, 3, 4, 5};
```

This is still not very powerful, but it will come in handy in some situations, such as the code examples in this document.

For more complicated cases, the library also provides a more verbose, but also more powerful feature that works with `LINQ` syntax.

```
var range =  Enumerable.Range(0, 10000);

var list   = from num in range.BuildFlexList()
             where predicate(num)
             select selector(num);

var vector = from num in range.BuildVector()
             where predicate(num)
             select selector(num);
```

The result of the `Build` methods is a type of `IEnumerable<T>` that implicitly generates a collection of the appropriate type. It's called a collection

builder, and it supports almost all LINQ operators and methods that have to do with transforming collections.

```
var range = Enumerable.Range(0, 10000);
var builder = range.BuildFlexList();
var list1 = builder.Take(100).Select(x => 2 * x);
var list2 = builder.Skip(10).Take(5).Where(x => x % 2 == 0);
```

Just like most other LINQ-type queries, a collection builder uses lazy evaluation. Unlike most other LINQ-type queries, it also caches the result so that it never needs to generate the same collection more than once.

Note that you must have a `using` reference to the namespace `Solid.Builders` in order to access the extension methods for creating a collection builder.

The builder constructs the collection incrementally whenever you iterate over it, when evaluation is explicitly forced using the `Force` method, or when you use an implicit conversion. Here are all the methods you can get the underlying collection:

```
var builder = Enumerable.Range(0, 10000).BuildVector();
var vector1 = builder.Force(); //explicit call
var vector2 = (Vector<int>)builder; //explicit conversion
var vector3 = builder; //implicit conversion
```

5 F# Interface

The library is written in C#, but it offers many features that target F# as well. There are located in a separate, companion assembly. This second assembly requires .NET Framework 3.5 and the core F# library.

5.1 Module interface

Each collection has a fully featured F# module, offering all the operations that don't have an operator alias (see Section 5.3) as `let` bindings. .

```
let list = XList.ofSeq {0 .. 1000}
           |> XList.select (fun x -> x + 1)
           |> XList.filter (fun x -> x % 2 = 0)
```

5.2 F# Overloads

The F# companion library adds extension overloads for every function which is partially incompatible with C#, such as functions taking a delegate parameter. Here is an example.

```
let list = XList.ofSeq {0 .. 10000}
let f1 = fun v -> 2 * v
let f2 = fun v -> v % 2 = 0
let f3 = fun a b -> a + b
let list2 = list.Select(f1)
                .Where (f2)
                .Fold  (0, f3)
```

These are more or less aliases for the same operations. They're still very comfortable to use in F#.

The reason I chose such an example, is that a lambda can bind to any delegate type. However, once you assign the lambda to a variable, in F# it will automatically bind to the type FSharpFunc. After this happens, it is no longer compatible with other delegate types. [1]

5.3 Operators

All collections support additional operators that act as aliases for instance-level methods. Here is a brief list of all the defined operators. Operator-like expressions are also included.

Some of these are defined as extension members for the type, others as inline let bindings.

Operator	Alias For	Usage	Available for
+>	AddFirst	4 +> target	XList
<+	AddLast	target <+ 4	XList,Vector
++>	AddFirstRange	[0; 1; 2] ++> target	XList
<++	AddLastRange	target <++ [0; 1; 2]	XList,Vector
<+>	AddLastList	target1 <+> target2	XList

I may add additional operators in the future, though.

5.4 Collection comprehensions

F#, unlike C#, supports custom collection expressions using computation expressions. Naturally, Solid collections support this capability. Here is an example of how it works, beside existing collection comprehensions.

```
let test1 = xlist  { for i in {0 .. 100} -> i }
let test2 = vector { for i in {0 .. 100} -> i }
let test3 = seq    {0 .. 100}
let test4 = query  { for i in {0 .. 100} do select 2 * i }
```

More complicated logic can be added as well.

[1] This means that sometimes, it will be difficult to tell whether or not a call passed through the extension method or went directly to the instance method. However, since the extension methods do nothing but call the instance methods themselves, it should never be relevant.

5.5 Active patterns

`Solid` collections also support consuming the collections using active patterns. Due to technical limitations that I haven't been able to overcome, the patterns are unfortunately partial patterns and will raise a warning unless a catch-all clause is added. It's also possible to suppress the warning explicitly.

The good thing is that each active pattern works for any collection that supports it. For example, all `Last` patterns are supported by both `Vector` and `FlexibleList`, and so you use the exact same pattern to access either. The same goes for the `Nil` pattern, which obviously all collections support.

```
let consumeLast col =
    match col with
    | Nil -> "Zero"
    | Last2(tail, item1, item2) -> "Two or more"
    | Last(tail, item1) -> "One or more"


let consumeFirst col =
    match col with
    | Nil -> "Zero" //The same pattern as the previous example.
    | First2(item1,item2,tail) -> "Two or more"
    | First(item1, tail) -> "One or more"
    | MatchList 3 [a; b; c] ->
        "Converts to a list. " +
        "The parameter denotes maximum length."
```

6 How Specific Operations are Handled

Here I'd like to provide some implementation details about specific operations. This should be incorporated into the reference API documentation.

6.1 Addition to the beginning

When a sequence is added to the beginning of a collection, it will be "joined" to its beginning, so that the first item of the sequence is the first item of the collection. Although this is not the most direct way to do this process, it is I think the most intuitive one.

6.2 Insertion

In the context of this library, *insert* always refers to insertion at index.

The result of the insertion is that the inserted item occupies the specified index. The item previously occupying this index, along with all proceeding items, is shifted forward by the appropriate amount.

When a sequence is inserted, it is "spliced" into the collection. The first item of the sequence occupies the specified index. It is proceeded by the rest of the items in the sequence, and finally by the previous occupant of the index and the rest of the items in the collection.

Consequentially, when the index is zero, insertion works the same way as addition to the beginning. When the index would place the item immediately after the collection, insertion works the same way as addition to the end.

You can insert into an empty collection as long as it is inserted at index 0.

6.3 Removal

In the context of the library, *remove* always refers to removal at index.

When an item is removed from the collection, the result is that it no longer occupies the collection. The indexes of all proceeding items are shifted back by the appropriate amount.

6.4 Empty collections

Empty collections don't support some operations that are clearly invalid for them. When an index is involved, the collection will throw an `index out of range` exception. When an index is not involved, such as when invoking the `First` property, an `invalid operation exception` is thrown instead.

Members that iterate over items in the collection, including methods like `TakeWhile`, work on empty collections. The result is usually an empty collection.

7 Not Done Yet

There are many more features I'm thinking of adding, but I haven't gotten around to them yet, or possibly I haven't decided how to do them. If anyone has any suggestions, I would love to hear them.

These are in no particular order.

1. The collections don't have proper structural equality and comparison yet. This is something absolutely necessary and a lot of the code is already there. I just haven't decided how to do it yet. It seems a tricky issue.
2. I'm going to add two more data structures: a key-value map, and a set. The key-value map exists, but I need to straighten things out first.
3. Possibly some more `C#`-specific features. Thing is, `C#` is a lot less flexible when it comes to that sort of thing than `F#` is.
4. I want to integrate collection builders discussed in Section 4.1 and the functional operations discussed in Section 3.5.

5. Currently, method, class, and interface names appearing in this documented are colored. I also want to make them link to the API and the MSDN where appropriate. I'm also thinking about changing the way these things are highlighted because it causes too many problems.
6. Put some real, reliable, and accurate performance figures here and possibly somewhere in the API.
7. Write up an implementation section. Will most likely need to rewrite some bits of the code so they're shorter and clearer. Been looking at Scala for this purpose.
8. Fix any inconsistencies between this document and the library.