Design Patterns for Lazy Evaluation

Dung ("Zung") Nguyen
Dept. of Computer Science
Rice University
Houston, TX 77251
dxnguyen@cs.rice.edu

Stephen B. Wong
Computer Science Program
Oberlin College
Oberlin, OH 44074
stephen.wong@oberlin.edu

Abstract

We propose an object-oriented (OO) formulation and implementation of lazy/delayed evaluation by reusing and extending an existing linear recursive structure (LRS) framework with the help of the strategy, decorator and factory design patterns. The result is a robust, flexible framework that can handle both infinite and finite lists and to which existing algorithms for finite lists can be applied without modification. The OO techniques used to develop this model are effective tools for teaching abstraction and design of data structures.

1 Introduction

Lazy/delayed evaluation is a technique used to represent large or infinite sequences ("streams") by not evaluating data terms until they are actually needed by the client code. In current teaching, lazy evaluation is often discussed in the context of functional programming languages such as Scheme [1]. Though streams have proven to be a useful and important concept, data structure texts in non-functional languages such as Pascal, C/C++, Smalltalk, and Java, avoid the topic altogether. This omission gives the false impression that lazy evaluation is relegated to just the functional programming domain.

We seek to dispel this misconception by presenting a formulation and an implementation of lazy evaluation that is based solely on the fundamentals of object-orientation: encapsulation, inheritance, and polymorphism. The implementation we propose here is an extension of the LRS framework provided by Nguyen and Wong [3]. It demonstrates the power of abstraction and how abstraction can be captured through design patterns. And, while illustrated using Java, our design solution can be implemented in any language that supports OO concepts.

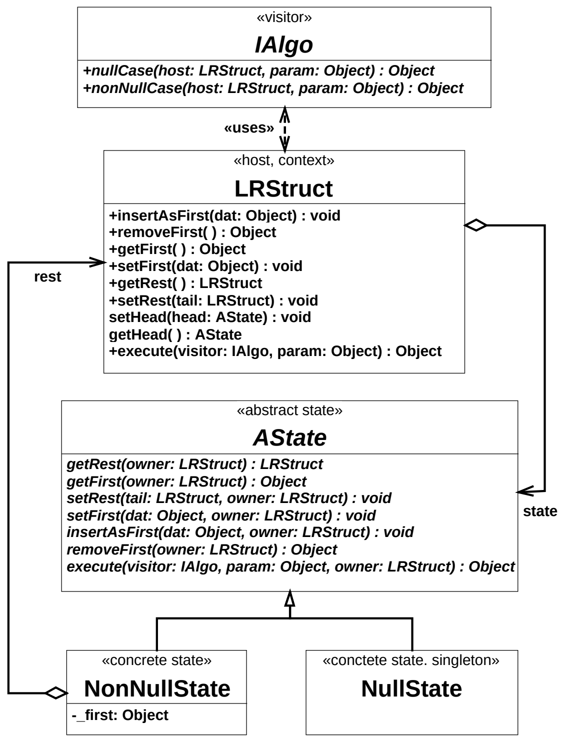
«visitor»
IAlgo
+nullCase(host: LRStruct, param: Object) : Object
+nonNullCase(host: LRStruct, param: Object) : Object
«uses»
«host, context»
LRStruct
+insertAsFirst(dat: Object) : void
+removeFirst( ) : Object
+getFirst( ) : Object
+setFirst(dat: Object) : void
+getRest( ) : LRStruct
+setRest(tail: LRStruct) : void
setHead(head: AState) : void
getHead( ) : AState
+execute(visitor: IAlgo, param: Object) : Object
rest
«abstract state»
AState
getRest(owner: LRStruct) : LRStruct
getFirst(owner: LRStruct) : Object
setRest(tail: LRStruct, owner: LRStruct) : void
setFirst(dat: Object, owner: LRStruct) : void
insertAsFirst(dat: Object, owner: LRStruct) : void
removeFirst(owner: LRStruct) : Object
execute(visitor: IAlgo, param: Object, owner: LRStruct) : Object
state
«concrete state»
NonNullState
-_first: Object
«conctete state. singleton»
NullState

Figure 1: The LRStruct framework.

Figure 1 shows Nguyen and Wong's LRS framework [3] in UML notation. We have taken the liberty of renaming the interface and some of the classes and their methods without modifying any of their semantics. At the heart of this framework are two design patterns:

- The state pattern to abstract and unify the different primitive behaviors of an empty list (NullState) and a non-empty list (NonNullState). The LRStruct forwards all requests to its current state, and behaves as if it changes classes dynamically.
- The visitor pattern to decouple the extrinsic algorithms performed on the LRS from the intrinsic structural behaviors of the LRS. Algorithms are implemented as visitors to the LRS and communicate with the LRS host via the host's public methods only. The LRS host has a public hook method to execute any conforming visiting algorithms.

Infinite lists differ from finite lists only in that the null case is never encountered. Conceptually, only the running time and memory requirements of a finite list algorithm should be affected when applied to an infinite list. Also, since lazy evaluation is a very useful technique for representing very large, but still finite data sets, the distinction between “lazy” lists and “eager” lists should be transparent, that is, they are abstractly equivalent. Thus, from our standpoint, if we are to extend this framework to add the lazy evaluation capability, we need only and should only extend the structural design of the LRS leaving its public behaviors and the interface with its algorithm visitors intact.

This paper shows how this extension is accomplished through the use of well-known design patterns [2]. The design process described here can serve as a non-trivial example illustrating to students how proper abstraction will enhance extensibility and reduce code complexity, and how design patterns help formulate the abstractions and implement them in an effective and elegant way.

Section 2 describes our design goals and their rationales. Section 3 formulates lazy evaluation as a “strategy” using the strategy pattern. We then combine the factory method pattern with the abstract strategy to manufacture lazy lists. Section 4 shows how the decorator pattern is applied to dynamically add and remove the lazy evaluation capability to the NonNullState (in Figure 1). Section 5 completes the design process by creating a factory class to manufacture eager and lazy lists and hide all implementation details of an LRS from client codes. Section 6 illustrates our OO implementation of lazy lists with several examples that lead up to the well-known Sieve of Eratosthenes algorithm to produce primes.

2 Design Considerations

We wish to impress upon students that good software design creates code that is reusable and extensible and that the key to creating good design is proper abstraction and decomposition. The process of adding lazy evaluation capability to the LRS framework in Figure 1 should serve as a test for the framework’s flexibility (ease of modification), and extensibility (ease of extension). It can also serve as an illustration for proper code reuse. In light of this, we have the following self-imposed design criteria. Lazy evaluation should be added to the given LRS framework in such a way that:

(a) All existing code need not be modified nor recompiled.

(b) All existing algorithms on ordinary finite lists should behave correctly on the extended framework. This is consistent with the abstract view that infinite lists, lazily evaluated finite lists, and ordinary finite lists are simply linear recursive structures.

(c) Whenever a LRS is created, it should appear to its clients as if it already contains what it is supposed to contain at the time of creation, irrespective of whether or not it is being lazy evaluated. As a consequence, its client need not be concerned whether or not the evaluation of the underlying data structure is delayed and should be able to manipulate it as if it exists in its totality. For example, suppose a client executes an algorithm on an infinite Fibonacci list to remove the third element and follows that with an algorithm on that same list to retrieve the 529th element. One would expect to obtain the same results as when the same algorithms are executed on a bounded Fibonacci list of sufficient length. This requirement is to support the view that a data structure need not and should not care about the data it contains. The only responsibility the structure has is to behave as advertised in its public methods. The responsibility to put the appropriate data in the structure lies with some other client of the structure but not with the structure itself.

(d) For a lazy LRS, the portion of the structure that has already been evaluated should have the same performance as an eager LRS, solely for efficiency purposes.

The first step toward achieving the above design goals is to define a class with lazy evaluation capability, say LazyLRStruct, by way of extending LRStruct. In the OO paradigm, there are two ways to extend a class: by composition or by inheritance. We rule out composition because the above constraints require that the new class must appear as a LRStruct to its clients, in particular to all existing algorithm visitors. As a result, we are left with defining LazyLRStruct as a subclass of LRStruct. By keeping all public methods of this class identical to those of LRStruct, its superclass, we make a LazyLRStruct appear to all clients simply as a LRStruct.

In order to understand how to construct a LazyLRStruct and how it behaves, we need to analyze what lazy evaluation entails, then identify and encapsulate the party that should be responsible for this task.

3 Lazy Evaluation as a Strategy and a Factory

Since an empty LRS needs no lazy evaluation for the obvious reason that there is nothing to evaluate, a LazyLRStruct, say L, must be non-empty by default. L contains a data element, called first, and, conceptually, another LRStruct called rest, which may be another LazyLRStruct. Lazy evaluation postpones computing L.rest until L.rest is needed. Upon inspection of the behavior of an eager LRStruct, only the calls L.getRest(), L.setRest(), and L.removeFirst() need to force the lazy evaluation mechanism to compute L.rest. For a LazyLRStruct that represents an arithmetic progression, lazy evaluation will return a LRStruct whose first is the next element in the progression. For a LazyLRStruct that represents the infinite sequence of primes sorted in increasing order, lazy evaluation will produce a LRStruct whose first is the next prime. Thus, lazy evaluation is an abstract algorithm that embodies the infinitely many ways to compute the rest of a LazyLRStruct,

and, in the pattern language, is said to be a "strategy" for a LRStruct, the "context," to compute its rest.
Figure 2 illustrates the strategy pattern [2]. In this pattern, lazy evaluation is represented as an abstract class (or a Java

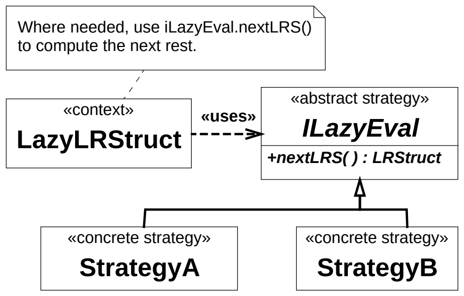

Figure 2: The strategy design pattern.

interface) called ILazyEval with an abstract method called nextLRS to compute and return a LRStruct, which may or may not be a LazyLRStruct. Thus both finite and infinite lists can be generated. The LazyLRStruct doesn't care how the rest is calculated, only that it can be calculated. The job of actually calculating the rest is delegated to a concrete ILazyEval strategy. Each specific lazy evaluation algorithm is encapsulated in a concrete subclass of ILazyEval. The concrete strategy is passed to the LazyLRStruct during its construction.

Since a LazyLRStruct must appear as a LRStruct to all clients, common OO design practices of information hiding suggest the use of the factory method pattern [2] to manufacture LazyLRStruct objects instead of allowing clients to directly use its constructor. This hides the details of the specific subclass construction from the clients. A factory method would instantiate a LazyLRStruct with the correct concrete ILazyEval installed. Since the ILazyEval already knows how to do this with its nextLRS() method, it is the natural candidate for the responsibility to create its own context. Thus we add the abstract method makeLRS() to ILazyEval for each concrete implementation of ILazyEval to manufacture an appropriate LazyLRStruct with itself installed as the lazy evaluation strategy. The ILazyEval interface is now:

```
interface ILazyEval {
   LRStruct makeLRS();
   LRStruct nextLRS();
}
```

In retrospect, we see that ILazyEval is a factory for inductively manufacturing LRStruct objects, where makeLRS() is the base case construction and nextLRS() is the inductive case construction. The details of how makeLRS() is implemented will be gradually made clear in the next three sections.

4 Decorating the NonNullState object

The state pattern of the existing framework suggests the addition of a new "lazy" state, say LazyNonNullState, for LazyLRStruct because all operations are delegated to the internal state. When a LazyLRStruct, L, is instantiated, its state is a LazyNonNullState object. Since L is non-empty by default, a LazyNonNullState object must appear to its clients as a NonNullState object for all existing legacy code to work properly. A safe way to achieve this is to hold an instance of a NonNullState and delegate calls to it. Since L forwards all requests to its current lazy state, should a request require L.rest, the lazy state needs to hold a concrete ILazyEval strategy and use it to compute L.rest. Moreover, once L.rest is computed, L should become eager in order to satisfy the performance requirement, and thus its state must become a NonNullState object. We thus need to have the capability to dynamically add/remove responsibilities to/from the NonNullState object without disturbing the existing framework. The decorator pattern [2] can help us achieve this design goal. The decoration is transparent to all clients and enables the desired state transition from lazy to eager. Figure 3 illustrates the decorator pattern applied to NonNullState.

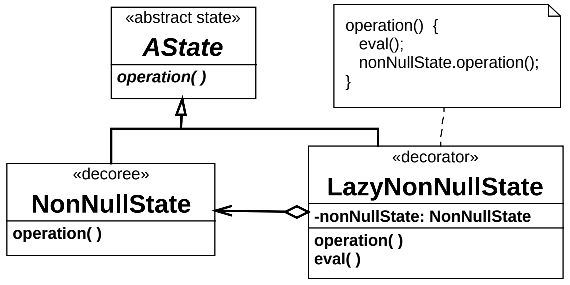

Figure 3: The decorator design pattern.

We define the LazyNonNullState as follows.

```
class LazyNonNullState extends AState {
  ILazyEval iLazyEval;          // the strategy.
  NonNullState nonNullState;  // the decorated component.

  LRStruct getRest(LRStruct owner) {
    eval (owner);
    return (nonNullState.getRest (owner));
  }

  private void eval(LRStruct owner) {
    owner.setHead (nonNullState);          // owner becomes eager.
    owner.setRest (iLazyEval.nextLRS ()); // owner.rest is computed.
  }
// constructor and other methods omitted
}
```

As prescribed by the decorator pattern, all the primitive methods of a LRStruct that need to access rest (getRest(), setRest() and removeFirst()) first prepare the owner LRStruct with eval(), and then forward the request to the enclosed NonNullState object to perform the actual operation. LazyNonNullState.eval() first switches the owner to its eager NonNullState state, and then calls on its lazy evaluation strategy to compute the next LRStruct, making it the rest of the owner. The methods that do not require rest need not be decorated and are simply forwarded to the enclosed NonNullState component. Figure 4 depicts a snapshot of the dynamic behavior of the decorating LazyNonNullState as it sheds its decoration and computes the LRS's rest.

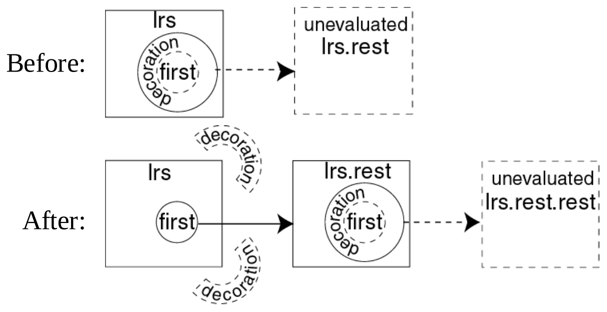

Figure 4: A lazy LRS before and after the evaluation of its rest.

5 Factory for LRStruct objects

Upon reviewing our current design, we see the role of the LazyLRStruct class strictly as a means to properly construct a LRStruct with a LazyNonNullState as its current state, and with the appropriate ILazyEval and initial value installed. Other than this, LazyLRStruct adds no behaviors to a LRStruct. Indeed, the clients do not care about the LazyLRStruct class; they only want to see LRStruct. Since the concrete ILazyEval classes are external to the list by virtue of being strategies, they should not be burdened with the responsibilities of the construction details of a LazyLRStruct, other than to provide the initial value and a lazy evaluation strategy. To this end, we create a factory class called LRStructFactory that is capable of instantiating a LazyNonNullState object with an initial value and a concrete ILazyEval strategy, and installing the resulting decorated NonNullState component as the initial state of a LRStruct. As a consequence, the LazyLRStruct class is no longer needed and can be removed from the design. By enclosing LRStructFactory in the same package with LRStruct, LRStructFactory will have the necessary and sufficient access to the relevant methods and constructors of the relevant classes to carry out its manufacturing task.

LRStructFactory is defined as a singleton and has two factory methods. One method creates eager (empty) LRStruct objects and the other creates lazy LRStruct objects with a given initial value and a concrete lazy evaluation strategy:

```
package LRStructure;
import LazyLRSEvaluators.ILazyEval;

public class LRSFactory {
  // singleton pattern code omitted
  public LRStruct makeLRS()  { return new LRStruct (); }

   public LRStruct makeLRS(Object initDat, ILazyEval lazyEval)  {
       LRStruct llrs = new LRStruct ();
       llrs.setHead (new LazyNonNullState (initDat, lazyEval));
       return (llrs);
   }}
```

In the end, the construction of a lazy list is a double layer of factories that demonstrates information hiding via proper levels of encapsulation.

6 Examples

To create a lazy list, all we need is to create a concrete ILazyEval and hand it to the LRSFactory singleton. Here we present examples of lazy evaluators that will generate an arithmetic progression, a list which filters all multiples of an integer from a given source list, and an infinite list of prime numbers using the Sieve of Eratosthenes algorithm.

A lazy evaluator that creates an infinite arithmetic progression (LazyIncEval) is straightforward:

```
public class LazyIncEval implements ILazyEval {
  private int val , increment;

// constructor code omitted.

  public final LRStruct nextLRS()  {
    val += increment;
    return makeLazyLRS ();
  }

  public final LRStruct makeLazyLRS()  {
    return LRSFactory.Singleton().makeLRS (new Integer(val), this);
  }}
```

A filtering lazy evaluator (LazyFilterEval) removes multiples of a given factor from a given source list (src), which may be finite, infinite, eager, or lazy.

```
public class LazyFilterEval implements ILazyEval {
  private Integer factor;
  private LRStruct src;  // The list from which all multiples of factor are removed.

  // constructor code omitted.

  public LRStruct nextLRS()  { return makeLazyLRS ();}   // same as base case!

  public LRStruct makeLazyLRS()  {
     src = (LRStruct) src.execute (SkipLeadMuls.Singleton(), factor);
     if (((Boolean)src.execute(IsNull.Singleton(),null)).booleanValue())
        return src;  // end of src = empty list
    else {     // src.first is now the next non-multiple of factor in src.
        Object nextVal = src.removeFirst ();
        return LRSFactory.Singleton().makeLRS (nextVal, this);
    }}}
```

In LazyfilterEval.makeLazyLRS(), SkipLeadMuls and IsNull are ordinary IAlgo algorithms to skip leading multiples of an integer in a finite, eager list and to check for an empty list, respectively, and are re-used unmodified here. Thus if the source is finite, the filtered list will be finite. If the source list is infinite, the filtered list will be infinite in this case, though it could conceivably be terminated by another type of filter evaluator. We have implemented a more sophisticated OO design of the lazy filter evaluator using predicate objects as strategies for filtering. However, space constraints prevent us from presenting it here.

We conclude with the classic Sieve of Eratosthenes (LazySieveEval). Our sieve works by making use of filters of type LazyFilterEval shown above, and by layering filters upon filters, using the latest value returned by the stacked filters to create the next one.

```
public class LazySieveEval implements ILazyEval {
  private LazyFilterEval lazyFilterEval;
  private Integer newVal = new Integer(2);

  public LazySieveEval()  {
   LRStruct src = (new LazyIncEval (2, 1)).makeLazyLRS (); // src=2, 3, 4, 5, ...
    lazyFilterEval = new LazyFilterEval (newVal, src);  // filter out even #s
  }

  public LRStruct nextLRS() {
   newVal = (Integer)lazyFilterEval.nextLRS().getFirst();
   lazyFilterEval = new LazyFilterEval (newVal, lazyFilterEval.makeLazyLRS ());
   return makeLazyLRS (); }

  public LRStruct makeLazyLRS()  {
   return LRSFactory.Singleton().makeLRS (newVal, this);
  }}
```

7 Conclusion

We have illustrated the process of applying design patterns to formulate and add lazy evaluation to an existing LRS framework without modifying and recompiling any of the existing code. The extension is illustrated in Figure 5. In short, we needed only add one class, LazyNonNullState, to decorate an existing class, and one interface, ILazyEval, to represent the strategy for lazy evaluation. A factory class, LRSFactory, was created for the purpose of information hiding only. The key to this extension process was the proper abstraction of the problem. We studied lazy evaluation in terms of its abstract relationship to a LRS. Proper abstractions combined with sound software engineering practices naturally lead to robust, extensible decompositions. We related those decompositions to standard design patterns and recognized that lazy evaluation could be viewed as an abstract strategy, ILazyEval, for the NonNullState component of a LRStruct. Adding a factory method to ILazyEval for the purpose of information hiding, brought out the inductive list building nature of the lazy evaluation process. Robustness and performance considerations induced the usage of the decorator pattern to wrap the NonNullState component, and then dynamically shed the lazy evaluation responsibilities after the next rest had been calculated. Finally, further information hiding considerations lead to encapsulating the entire LRS framework within a factory, including the lazy evaluation, and the elimination of an unnecessary class (LazyLRStruct).

As illustrated in the preceding sections, the coding for the extended framework and for many of the concrete lazy evaluators is quite simple. This is a result of having incorporated the invariant behaviors of lazy evaluation into the LRS framework while encapsulating and separating the variant portions as an abstract lazy strategy. Whether or not a student has seen a functional implementation of lazy evaluation, this OO version enhances his/her understanding and appreciation of key OO concepts such as inheritance, polymorphism, and design patterns. It also enhances the student's understanding of programming in general by providing an alternative viewpoint into a topic presented in another context. The process illustrated in this paper entails repeated abstraction and decomposition. It helps the student develop his/her problem solving skills because it is an iterative process driven by the search for the abstraction that best models the problem and simplifies the solution.

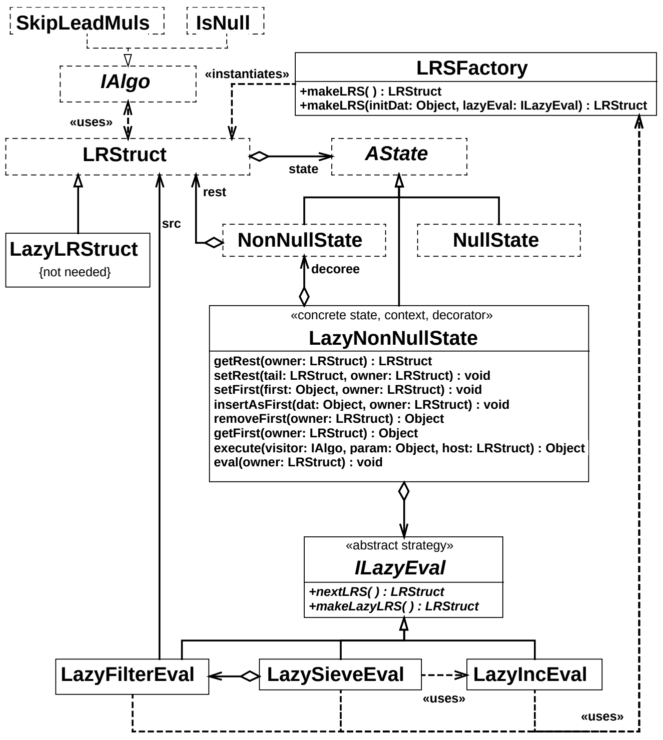
SkipLeadMuls
IsNull
IAlgo
«instantiates»
LRSFactory
+makeLRS( ) : LRStruct
+makeLRS(initDat: Object, lazyEval: ILazyEval) : LRStruct
«uses»
LRStruct
state
AState
rest
src
LazyLRStruct
{not needed}
NonNullState
NullState
decoree
«concrete state, context, decorator»
LazyNonNullState
getRest(owner: LRStruct) : LRStruct
setRest(tail: LRStruct, owner: LRStruct) : void
setFirst(first: Object, owner: LRStruct) : void
insertAsFirst(dat: Object, owner: LRStruct) : void
removeFirst(owner: LRStruct) : Object
getFirst(owner: LRStruct) : Object
execute(visitor: IAlgo, param: Object, host: LRStruct) : Object
eval(owner: LRStruct) : void
«abstract strategy»
ILazyEval
+nextLRS( ) : LRStruct
+makeLazyLRS( ) : LRStruct
LazyFilterEval
LazySieveEval
LazyIncEval
«uses»
«uses»

Figure 5: The Lazy LRS framework (original classes outlined in dashed lines).

Acknowledgment

We thank Robert "Corky" Cartwright who shared his Java code for the Sieve of Eratosthenes using functional Scheme-like style and inspired us to write this paper.

References

[1] Abelson, H., and Sussman, G. *Structure and Interpretation of Computer Programs*, 2nd ed., MIT Press, 1996.

[2] Gamma, E, Helm, R, Johnson, R, and Vlissides, J. *Design Patterns, Elements Of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[3] Nguyen, D. and Wong, S. *Design Patterns for Decoupling Data Structures and Algorithms*, SIGCSE Bulletin, 31, 1, March 1999, 87-91.