LaVaZza - Laziness for Java and Pizza

Marcin Benke Rafał Błędziński
Institute of Informatics
Warsaw University

July, 1999

Abstract

Pizza [OW97] is a conservative extension of Java bringing some important features from the functional world. It has gained substantial popularity[1]. Nevertheless it lacks a feature dear to many functional programmers: lazy evaluation. We claim that this gap can be filled in an elegant, efficient, compatible and orthogonal way. We describe a concept of type-driven laziness which has allowed us to design and implement Lazy Pizza.

1 Introduction

Java [GJS96] is a strongly typed object oriented language. Its definition requires that implementations rely on heap allocation with garbage collection. It also ensures safe execution that never corrupts neither the stack nor objects on the heap. Besides, a great advantage of Java is its portability and popularity in the Internet.

All of the above suggest sticking to the principle that the efforts towards extension of Java should be done while preserving full back compatibility, in order not to lose access to existing libraries and software. Pizza [OW97] is a language which was created with this principle in mind. It is a strict superset of Java. It incorporates the following ideas to Java:

- parametric polymorphism
- higher-order functions
- algebraic data types

Being an imperative language, Java supports eager evaluation only. Pizza, though being a huge step towards the functional world, does not change this. So here comes a question that may seem absurd to "object-oriented" programmers, yet obvious for the "functional" ones: why not make it lazy? Or — as

[1]Pizza was honored with the Gamelan "cool" award.

Pizza does merge functional and imperative features, try to merge eager and lazy evaluation?

Sometimes it is desirable that some parts of the code should be performed in a lazy fashion (just like in Haskell). It is especially useful with lazy data structures (e.g. lists). Of course one can emulate lazy evaluation manually, but in all but trivial cases it very soon becomes messy and is difficult to implement in a transparent fashion.

So we asked ourselves a question: is it possible to introduce elements of laziness to Pizza (or Java) in similar way that Pizza augmented Java with functional elements? This may (or even should) be of course viewed as a part of a more general question about introducing laziness elements to arbitrary object oriented language: its expediency, gains vs. costs, utility of the solutions.

An ideal solution should possess following properties:

1. **Elegance:** small (or no) changes in language grammar (it's not a completely new language, but the same with some new properties).

2. **Ease of use:** from the programmer's point of view lazy objects have the same properties as corresponding eager objects:

   - their definition should not differ from this of eager objects;
   - lazy objects may appear in all contexts where corresponding eager objects may.

3. **Compatibility:**

   - all existing source code in Pizza and Java is accepted to be Lazy Pizza and has the same semantics (Lazy Pizza is a superset of Pizza).
   - all compiled libraries and existing programs written in Pizza and Java should be easy to use in Lazy Pizza without recompilation.
   - no extensions to the virtual machine: the Lazy Pizza compiler should eventually produce code that runs on stock JVM (or indeed Java source code)

4. **Efficiency:** no overhead for the strict parts of the code and as little overhead as possible for the lazy parts.

5. **Orthogonality:** independence from the original Pizza extensions i.e. the solution should yield Lazy Java when applied to pure Java.

1.1 The concept of "lazy types"

The idea of "partial laziness" may seem controversial, but let us recall that it is not really new: in the language Hope [BMS80] all type constructors are lazy but other functions are strict. This reflects the observation that the place where laziness is most needed is synthesis and analysis of complex data objects (as opposed to, for example, arithmetical operations, which should indeed be strict).

Let us analyze possible ways of introducing laziness to Java and Pizza:

1. one can make all manipulations on objects become lazy, leaving operations on simple objects (numbers, characters) strict. This however would violate the principle of compatibility, since old libraries (including standard ones!) would have to be recompiled.

2. one can restrict laziness only to the functional extensions introduced by Pizza. This however contradicts our desire for orthogonality and does not seem flexible enough.

3. one can leave the choice of which classes should be subject to lazy evaluation to the programmer.

The last approach looks the most promising so we choose to pursue it. The next section explores the design space in more detail and explain the reasons for our choice.

The syntax is augmented with new keyword `lazy`. It can occur only as a class definition modifier. The definition `lazy class` C specifies that objects of class C and its subclasses are to be evaluated in a lazy fashion. More precisely, *all expressions* of type C are evaluated lazily. There is an additional restriction: a lazy class can inherit only from class *Object* or from another lazy class. We shall explain the reasons for that in due course.

All computations in lazy types are lazy. Specifically, the following operations are delayed:

- lazy object construction

- calls to functions and methods with a lazy result type.

According to the lazy evaluation paradigm, a suspended computation is evaluated when it's result is needed (and then never reevaluated). However, as we shall see further there several possibilities of defining the notion "needed", which lead to different implementation strategies.

Before we delve into details, let us observe that generally a lazy object should be forced when

- one of its elements (method, attribute) is needed, or

- it is cast to a non-lazy type.[2]

[2] In our implementation casts of a lazy type to a non-lazy type other than *Object* cannot occur, but we list this case for the sake of completeness.

2 Exploring the design space

Let us start by taking a look at the problem of introducing laziness into object-oriented languages in general. We shall analyze several potential solutions and evaluate them based on their usefulness and implementation costs. Since our primary aim is to extend Pizza and Java, all examples shall be drawn from these languages, but most of the analysis applies to any object-oriented language of a similar genre.

Further on, we shall describe chosen solution in more detail.

2.1 Total vs partial laziness

The first question to be answered is thus: how much laziness do we want? Some possible answers are:

(a) Total (or global) laziness — all language constructs are subject to lazy evaluation; this solution is quite common solution among non-strict functional languages

(b) Partial (local) laziness — some computations are suspended until their result is needed; solutions of this kind come in two syntactic flavours:

   (i) explicit: evaluation suspension and/or forcing via syntactic constructs

   (ii) implicit: context analysis determines suspension and forcing points.

It seems that the global approach inevitably conflicts at least two of our goals: Compatibility (lazy evaluation of classes compiled with an "ordinary" Java or Pizza compiler doesn't seem possible without changes to JVM and even if it was, it might lead to changes in their behaviour) and Efficiency (lazy evaluation of classes that doesn't need it may incur undesirable overheads).

We should thus look at the local approach and examine its flavours in more detail. The explicit flavour has some merits: the programmer has full control over thunk construction and forcing. It is also the simpler solution that can be realized using but a macroprocessor.

Also the implicit flavour has some merits: it shifts the burden from the programmer to the compiler. Now, however the programmer has hardly any influence upon the choice of evaluation style. Worse still, compiler choices can vary from one implementation to another.

Bearing all this in mind, we choose a mixed solution that seems to combine merits of both flavours: some classes are explicitly marked as lazy and this marking leads to (via a formal set of rules) a code transformation performed by the compiler.

```
typedef LazyList = Nil |              -- []
                   Cons Int LazyList -- n:t

arithSeq n m = Cons n (nmlist (n+m) m) -- artihmetic sequence
odds = arithSeq 5 2 -- [5,7,..]
primes = Cons 2 primes1
notdiv (y:ys) x = if y*y<=x then ( ((mod x y) /= 0)
                                  && (notdiv ys))
                             else
                                True

primes1 = Cons 3 (filter (notdiv primes1) odds)
```

Figure 1: Infinite list of all primes — Haskell version

As an illustration let us consider the notorious example of non-strict evaluation: creation of an (infinite) list of all prime numbers. Its Haskell version could look[3] as depicted in 2.1.

2.2 Ways of introducing laziness

Let us in turn consider several possible approaches to laziness

Lazy methods: some method calls are suspended until their result is needed. An important design decision here is: which calls? There are many possible answers, including:

(i) all calls to methods explicitly marked lazy.

(ii) calls explicitly marked lazy (this is in fact an instance of the fully explicit flavour described earlier)

Either way however, it turns out that with this approach, achieving Compatibility (if at all possible to the full extent) needs very careful marshaling and leads to complicated and inefficient implementations.

Call by name is an interesting construct that is close to but in reality not the same as laziness (aka call by need). It allows to achieve something akin to lazy evaluation, but is too weak to implement infinite data structures.

[3]We definitely refrain from using special Haskell syntax for lists to facilitate comparison with Pizza.

Call by need some parameters are designated as being subject to lazy evaluation. For lists, we could designate head or tail (or both) to be lazy. Here's an example of how one could write a list of *int* with a lazy tail:

```
class List {
    case Nil;
    case Cons(int n,lazy List t);
...
}
```

In this case a thunk would be created at the time of list node creation: the evaluation of the second parameter of the constructor would be suspended till the first access to the list tail.

This method may be considered as a finer-grained version of lazy methods (or lazy constructors) but still does not seem satisfactory.

Lazy constructors In this solution, for a class marked by a programmer as lazy, all its constructors are delayed. This method seems already quite feasible. Note that it can be applied to pure Java as well as to Pizza. Here's how it can be used to implement lazy lists:

```
lazy class LazyList {
       case Nil;
       case Cons(int n,LazyList t);

static    LazyList  nlist(int n)
  {  return new Cons(n,nlist(n));  }

static    LazyList  nlist1(int n)
  {  LazyList l=nlist1(n);
     return new Cons(n,l);  }

...

}
```

The attentive reader will notice that there is an unpleasant difference between the two realisations of `nlist` above: the first one works as expected, creating a (lazy) infinite list, while the seconds leads to an endless recursion. The next method shows how to improve this.

2.2.1 Lazy types

This idea is a hybrid of two of the ideas analysed above: lazy methods and lazy constructors. A type may be designated as lazy; then all constructors of this type as well as methods returning values of this type are evaluated lazily. Every such suspended computation is executed at most once, when one of the following operations is performed on the yet-unevaluated thunk:

- attribute selection
- method selection
- cast to a non-lazy type.

As with lazy constructors, this method applies both to Java and Pizza. Figure 2.2.1 shows how to implement an infinite list of all primes using "lazy types".

Note that in this case the problem described earlier disappeared. Both `nlist` and `nlist1` execute correctly and return the same result (an infinite list).

This approach seems the most promising so we have chosen it as the basis for design and implementation of LaVaZza.

3 Lazy types in detail

Having decided upon the type-driven laziness, there are two issues one has to address: inheritance and interfaces. So far we have stated that a class defined with the modifier `lazy`, is a lazy type. The main questions we have to answer are:

Q1 Should the modifier `lazy` be inherited?

Q2 Should one allow lazy interfaces?

It seems that the answer to (Q1) should be positive: all subclasses of a lazy class should be lazy.

Regarding (Q2) we feel that lazy interfaces would have little sense, not to mention foggy semantics and difficult implementation. Therefore we decide to exclude them from our design. This decision also allows us to use interfaces as an elegant link between lazy and strict parts of program.

The decisions described above allow us to state that lazy types:

- are classes,
- form trees according to inheritance hierarchy

Having said that, one soon notices that this leave open one more question:

Q3 Can a lazy class have non-lazy superclasses other than *Object*?

```
lazy class LazyList {
       case Nil;
       case Cons(int n,LazyList t);

static    LazyList  nlist(int n)     // this works
  {  return new Cons(n,nlist(n));  }
static    LazyList  nlist1(int n)    // this works, too
  {  LazyList l=nlist1(n);
     return new Cons(n,l);  }
static    LazyList  arithSeq(int n,int m)
  {  return new Cons(n,arithSeq(n+m,m));  }

  static LazyList primes;
  static LazyList primes1;
static {
    primes1 = new Cons(3,countPrimes(nmlist(5,2)));
    primes = new Cons(1,new Cons(2,primes1));
       }
static LazyList countPrimes(LazyList  l)
{
    switch (l) {
      case Cons(int n,LazyList t):
                     if (notdiv(n))
                         return new Cons(n,countPrimes(t));
                     else
                         return countPrimes(t);
    }
    return null;
}
static boolean notdiv(int n)
{
    LazyList l = primes1;
    while (true) {
      switch (l) {
        case Cons(int n1,LazyList t):
                      if (n1*n1 > n)   return true;
                      if ((n % n1)==0) return false;
                      l=t;
                      break;
}}}}
```

Figure 2: Infinite list of all primes — Lazy Pizza version

One could argue for the positive answer to this question from a design point of view. However, it seems that such solution would be difficult to implement in an efficient (and sane) way. For example, it turns out that this approach severely limits the possibilities to implement infinite data structures.

Last but not least, it seems that the design we assume, where all lazy types form hierarchies inheriting directly from *Object* has much more comprehensible semantics.

4 Implementation concerns

It turns out that lazy types do not bring us many implementation concerns. It is sufficient to describe methods of closure representation, formation and forcing. We have considered and tested three approaches to closure representation.

4.1 Universal closures

This approach attempts to represent all closures as objects of one universal class *LazyClosure*.

Class hierarchy of the lazy classes remains almost unchanged in the translation process. Every class being a root of a lazy subtree is made to inherit from the special class *LazyObject* instead of *Object*.[4]. This allows for a very simple implementation — all laziness features are contained in *LazyObject* (closure formation) and *LazyClosure* (forcing). However, it turns out that this approach has one annoying misfeature: does not implement method overloading to different lazy types properly. This problem is efficiently solved by the following approach.

Program transformation: all calls to constructors and methods of a lazy result type are suspended yielding an appropriate closure. Access to object attributes or methods as well as a typecast to an interface, subtype or eager supertype forces a closure.

4.2 Corresponding closures

One solution to the problem above is to make closure hierarchy mirror the lazy class hierarchy: every lazy class has its corresponding closure class. This leads to some increase in a code size (but not very significant as the closure classes don't have much code), but the execution time remains the same as with the previous approach.

[4]Recall that a lazy class cannot have any eager ancestors other than *Object*

Class hierarchy: same as in previous method, with the addition of a closure hierarchy rooted at $LazyClosure$

Program transformation: as in the universal closures method.

Interfaces: in both of the above approaches, the closure classes do not implement interfaces. Thus, when passing a closure to an universal algorithm, the implicit typecast forces evaluation of a single node (the rest of the structure may safely remain unevaluated). We do not regard it as a substantial problem, but still, there is a solution that can guarantee even fuller laziness (alas, not without general overhead).

4.3 Interface closures

This approach extends corresponding closures in that every closure implements all the interfaces that the corresponding lazy class does. In addition, as previously any closure could be either unevaluated or evaluated, now a closure can be in one of four states:

(i) unevaluated,

(ii) partially evaluated — an indirection to another closure in state (iii) or (iv),

(iii) constructing — a suspended computation of a constructor

(iv) evaluated — the closure points to a computed object

This allows us to implement typecasts to interfaces and subclasses without forcing evaluation at the point of cast (of course casts to eager superclasses still require evaluation, but this is desired).

Class hierarchy: same as in the previous method, but the closure now implement all interfaces of their respective lazy classes.

Program transformation: the only substantial difference to the previous methods is the treatment of casts, which now do not require that the closure is forced.

4.4 Comparison of the implementation methods

It seems that the "corresponding closures" approach is the best: efficient and flexible enough. The awkward misfeature of the universal closure seems grave enough to disqualify it, and the costs of interface closures seem to outweigh their benefits. Yet we have implemented all methods and leave the choice to the user.

5 Conclusion

We hope to have proved that the crazy idea of adding laziness to Java or Pizza is not so crazy after all. The idea of type-driven laziness has led us to the design and implementation that are in our opinion quite elegant, fully compatible, relatively efficient and also orthogonal to the original Pizza extensions.

Of course, there is definitely much room for improvement and still many questions unanswered. We regard the design described above as a foundation and starting point for further development.

Would you like a cup of Lavazza?

References

[BMS80] R. M. Burstall, D. B. MacQueen, and D. T. Sannella. Hope: an experimental applicative language. In *Conference Record of the 1980 LISP Conference*, pages 136–143. ACM, ACM, August 1980.

[GJS96] James Gosling, Bill Joy, and Guy L. Steele. *The Java Language Specification.* The Java Series. Addison-Wesley, Reading, MA, USA, 1996.

[OW97] Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 146–159, Paris, France, 15–17 January 1997.

[OW98] Martin Odersky and Philip Wadler. Leftover curry and reheated pizza: How functional programming nourishes software reuse. In *Fifth International Conference on Software Reuse*, Vancouver, BC, June 1998. IEEE.