LazyJ: Seamless Lazy Evaluation in Java

Alessandro Warth

October 7, 2005

LazyJ\(^1\) is a backward compatible extension to the Java programming language that allows programmers to seamlessly tap into the power and expressiveness of lazy evaluation. It does so by extending Java’s type system with lazy types. A variable of type lazy T (where T is any Java type) can hold a thunk which when evaluated will yield a value of type T.

Other languages with strict semantics—including Scheme and O’Caml—provide support for lazy evaluation through delay and force functions which can be used by programmers to achieve a “manual” form of lazy evaluation. But programming using these functions can be quite onerous.

LazyJ’s raison d’être is its novel type system which includes lazy and eager types, and provides coercions between these types. Specifically, when the type checker finds an expression of a lazy type where an expression of a non-lazy type is expected, it automatically forces that expression. Conversely, when the type checker finds an expression of a non-lazy type where an expression of a lazy type is expected, it automatically delays that expression. Thanks to these coercions, programmers can write lazy code without ever having to worry about explicitly delaying or forceing expressions. In fact, LazyJ does not even provide explicit delay and force operations. The only new piece of syntax it adds to Java is the lazy type modifier. Consequently, LazyJ programs are a lot more readable and easier to understand than equivalent programs written with explicit delay and force operations.

And now for a little taste of LazyJ... One important benefit of lazy evaluation is that it can be used to describe infinite data structures. Given the following declarations

```java
class Node
{ int data;
  lazy Node next;
  Node(int d, lazy Node n) { data=d; next=n; }
}
Node intsFrom(int n) { return new Node(n, intsFrom(n+1)); }
Node filter(int n, Node l)
{ if (l.data%n==0) return filter(n, l.next);
  else return new Node(l.data, filter(n, l.next)); }
Node sieve(Node l) { return new Node(l.data, sieve(filter(l.data, l.next))); }
Node primes=sieve(intsFrom(2));
void printFirst(int n, Node l)
{ if (n==0) return;
  else { System.out.println(l.data); printFirst(n-1, l.next); }
}
```

the call printFirst(1000, primes) will do exactly what it suggests. Note that the only uses of the lazy keyword are in the declaration of the Node class. The rest of the code is written in a very straightforward manner, implicitly using lazy evaluation whereever appropriate.

\(^1\)http://www.cs.ucla.edu/~awarth/lazyj