RT Hatfield

Alan Moody

October 29th 2015

Laziness

Java :

See the included file, lazyCheck.java.  The second argument to stateChange

is never used, and so whatever expression might be supplied as that

argument in a call to that function should not be evaluated in the case

of lazy evaluation.  Instead, it would remain in thunk form for the life

of that part of the call stack.  Therefore, it is easy to use mutable

state (normally a lazy eval no-no, but available in Java) to check if

an expression has been evaluated or not.  In my example, b is initially 2.

If the call to stateChange(•) is evaluated lazily, ++b will never be

evaluated and b will still be 2 after that call, so "lazy" is printed.

Otherwise, ++b is evaluated, b becomes 3, and "eager" is evaluated.  Results

show that Java is, indeed, evaluated eagerly.

Strictness:

The change that Doug proposes in this situation makes the program “overly eager”. Included are files “Laziness\_1” (the original example) and “Laziness\_2” (Doug’s example). The results from the tests are indeed different, as you can see in the case:

(interp (parse '{with {y blah} y}) (mtEnv))

State and Lazy Evaluation:

When you evaluate lazily, you leave a lot of details up to the

language itself.  This can lead to weird situations where you

may not be entirely sure of when some bit of code will get evaluated,

and if you are programming statefully, you most likely depend on

evaluating in a particular order and modifying that state in a particular

order.  However, with lazy eval, your expressions may not get evaluated

exactly in the place that you wrote them, and then the state will

be modified in a different order.

a = 2; // initial state

define funcX (any) {

return any;

}

define mystery () {

if (rand > 0.5) {

return 1;

}

return 0;

}

funcX(a--); // a = 1

funcX(a--); // a = 0

funcX(mystery() && a--); // a-- may not be evaluated

funcX(a++ / a--); // a++ and a-- evaluated, but possible

// divide-by-zero

a = funcX(a); // no idea what a is now, completely

// nondeterministic

Application:

Understanding the small differences between lazy evaluation and eager evaluation can make a helpful difference in all sorts of programming practices. For example, lazy evaluation is helpful if I'm in a situation where I might be using a possibly infinitely-sized data set, or if due to work requirements I'm writing in a language like Haskell or Scheme.  Knowing more about eager evaluation is helpful when I'm writing code in Java or C++, because in those cases I can be more careful about, say, not passing a ton of items into a function that may not use them all (which is a way of looking at the single responsibility principle).

In general it is good practice while coding to make sure that all your branches are covered, and likewise that you are free from any sort of dead code. Programming with a lazy mind set can help to be aware of when your parameters are being evaluated, and when they actually aren’t being used much or at all. With this in mind, I can be more conscious of the parts of my code that are useful, and those that aren’t, and thus be a more efficient programmer.

As we mentioned above, languages that favor lazy programming (like Racket or Haskell) are easily the right choice for handling infinitely sized data sets. Being able to pass these around as promises without actually evaluating them can be very useful, and will keep your code running efficiently and fast.