Lazy evaluation in Haskell

James Thompson

By default Haskell uses lazy evaluation - this means that expressions are not evaluated until necessary, this even applies to expressions passed as arguments to functions. This can be very useful when doing tail recursion, since we do not have to evaluate anything until we reach the base step, reducing run time. The cost for this is can be increased and unpredictable memory usage.

In this example (`Problem5.hs`) we create 3 large lists of pseudo-random random integers (20 million elements). Then, for each list, we calculate the sum of the smallest 50 elements. Finally we print out a message depending on these sums.
Haskell's lazy evaluation allows it to make two timesavings over strict languages.

- Since the program only uses the first 50 elements of the sorted list, Haskell can bail out of the sorting algorithm as soon as those elements have been sorted, rather than having to sort the whole list. This does depend on the specific sort algorithm used, base Haskell uses quick sort (link) which tends to sort from smallest to largest so this shortcut is possible.

- The other short cut Haskell can take is in `comparer`. In this example the function exits at the first return statement so the arguments `v2` and `v3` are never evaluated, which means `value2 = sumSmallestN 50 ys` and `value3 = sumSmallestN 50 zs` are never evaluated in `wrapper`. And therefore `list2` and `list3` are never created since they are never used.

For comparison we provide the equivalent code in python and C++ (`Problem5.py` and `Problem5.cpp`). Some benchmark timing and memory usage information is shown below.

Haskell		C++		C++ with -O2		python	
time	memory	time	memory	time	memory	time	memory
3,25s	1173 MB	22.64s	472 MB	4.61s	472 MB	26.21s	2480 MB

While this is not necessarily a fair comparison, it is worth noting that with the `-O2` flag the C++ code is as fast as and uses less memory than the Haskell code. gcc's optimiser has probably taken some of the same short cuts that Haskell does naturally.

Haskell's large memory usage is a consequence of the purely functional nature of the language - whereas C++ and python can do the sorting in place, Haskell needs to create allocate a large amount of memory during the sorting algorithm since the original list is immutable. Recall Haskell never even created two of the three lists, so the fact is used more twice as much memory as C++, which created all three, is somewhat surprising.

Lazy evaluation combined with pureness means it can often be very diffiuclt to intuit the memory usage of Haskell compare to other low-level, compiled languages like C++ or Rust.

Document generated: 2021-02-25