Noname manuscript No.
(will be inserted by the editor)

An Investigation Into the Use of Haskell for Dynamic Programming

David McGillicuddy · Andrew J. Parkes · Henrik Nilsson

Received: date / Accepted: date

Abstract Over the last decade the speed of computers has increased by many orders of magnitude but the speed of the typical programmer has not. In many cases it is far more important to quickly produce correct and robust code than to optimise code for performance, and as computers continue to become more powerful, while humans will essentially remain the same this is ultimately going to become the norm. We argue that prototyping new heuristics and algorithms for combinatorial optimisation is one area where speed of development of correct code is already more important than absolute performance.

As an example, there was a high profile case recently[1] in which a software bug has caused erroneous results to be published, and to possibly influence European policy[1]. Amongst other things, a cause of error was a range-indexing mistake in the spreadsheet which caused several countries to be excluded from the analysis.

Consider the simplified example of summing a collection of numbers: in a high level programming language like Haskell one passes the name of a collection of numbers (whether an array, a list or otherwise) to the *sum* function which will, behind the scenes and opaque to the user, index each element and add them together thus completely eliminating that class of errors. In most spreadsheet software however one has to manually select the cells (e.g. "C3:C100")[2] which is error-prone as well as being non-trivial to later expand to include additional data.

D. McGillicuddy
University Of Nottingham
E-mail: dxm@cs.nott.ac.uk

A. J. Parkes
University Of Nottingham
E-mail: ajp@cs.nott.ac.uk

H. Nilsson
University Of Nottingham
E-mail: nhn@cs.nott.ac.uk

[1] `www.bbc.co.uk/news/magazine-22223190`

[2] While named ranges do exist they still have to be manually specified which just pushes the problem elsewhere.

TODO: talk about reasoning about code

We use Haskell to implement standard dynamic programming algorithms, including bounded and unbounded knapsack, and column generation. Then we compare with implementations in Java and C in terms of speed, conciseness, modularity, as well as ease of parallelisation, refactoring, debugging and reasoning. To make the comparisons fair we keep the structure of the code similar across languages, except when taking advantage of specific language features (e.g., pointers, objects or laziness). The implementations are idiomatic and representative of an 'average' user, without non-portable micro-optimisations. In particular, standard libraries are used throughout for data structures, mathematics and floating-point arithmetic with as little as possible implemented by hand.

Our preliminary results (unbounded knapsack in C and Haskell) show that while the C code is about four times faster, using Haskell for prototyping indeed offers significant advantages in terms of speed of development and eliminating certain classes of errors, without incurring a performance penalty that is unacceptable for a prototype.

Consider the following examples from the unbounded knapsack implementations to find the Greatest Common Divisor (gcd) of an array of n weights, $\mathbf{W}$, and the initial capacity c. The function gcd (which takes two integers and returns the largest integer which cleanly divides both of them) is associative, so

$$gcd(c, \mathbf{W}_0, \ldots, \mathbf{W}_{n-1}) = gcd(gcd(\cdots gcd(gcd(c, \mathbf{W}_0), \mathbf{W}_1)\cdots), \mathbf{W}_{n-1})$$

and the code needs to apply gcd pairwise to the capacity and each weight, reducing them to a single integer after n calls to gcd. The basic algorithm is demonstrated in the C implementation - there exists an accumulator variable gcd_all which is initialised to $capacity$ and then gcd'd with each weight. Note that it was necessary to manually specify the bounds of the loop, index each element separately and then update the accumulator variable manually with the result of gcd for each new $\mathbf{W}_i$.

Looping over each element of an array is such a common task that in Java it has been abstracted over with the *for-each* loop. This avoids the problem of having to manually specify the bounds of the loop or index into the array at the cost of a lack of flexibility - since the current index is not in scope as it would be inside a normal for loop, a for-each loop is not a good fit for code in which access to the index is required. In this case however, as shown in the Java implementation of gcds, this is not necessary so using a for-each loop reduces the boilerplate that the user needs to type and hence the number of places that an error can appear. The accumulator variable still needed to be manually updated in each iteration however.

In Haskell the idiom of updating an accumulator variable with each element of a list using a binary function is abstracted over using the function $foldl'$ defined by

$$foldl'(f, acc0, \mathbf{W}) = f(f(\cdots f(f(acc0, \mathbf{W}_0), \mathbf{W}_1)\cdots), \mathbf{W}_{n-1})$$

and so the Haskell definition of gcds calls $foldl'$ with the function gcd, the initial accumulator value of c and the vector $\mathbf{W}$ thus avoiding the need for the programmer to manually specify the range of the loop or how the accumulator should be updated.

TODO: benchmarking gcd isn't the whole story due to fusion etc

```
                gcd_all = gcd (gcd_all, weights[i]);
        }
        return gcd_all;
}
```

Fig. 1: C99

```
private int[] WS = {26,90,65,32,84,78,69,77,58,85};

public int gcds (int capacity, int[] weights) {
        int gcd_all = capacity;
        for (int weight : weights) {
                gcd_all = gcd (gcd_all, weight);
        }
```

Fig. 2: Java 7

```
gcds :: Int -> Vector Int -> Int
gcds capacity weights = foldl' gcd capacity weights
```

Fig. 3: Haskell

TODO: Graph of knapsack results here? gcc -O2 vs ghc-O2

Keywords Haskell · C · Java · Functional Programming · Dynamic Programming · Language Comparison

References

1. Herndon, T., Ash, M., Pollin, R.: Does high public debt consistently stifle economic growth? a critique of reinhart and rogoff. Cambridge Journal of Economics (2013). DOI 10.1093/cje/bet075. URL `http://cje.oxfordjournals.org/content/early/2013/12/17/cje.bet075.abstract`