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.

```c
int gcds (int capacity, size_t n, int weights[restrict static n]) {
        int i, ans = capacity;
        for (i = 0; i < n; i++) {
                ans = gcd (ans, weights[i]);
        }
        return ans;
}
```

Fig. 1: C99

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

Fig. 2: Java

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.

TODO: extend this, talk a little more about results and the actual code - compare loops vs folds and maps? This is where you convince people to read it

Consider the following example from the knapsack implementations to find the Greatest Common Divisor (gcd) of an array of n weights w_i and the initial capacity c. The function *gcd* (which takes two integers and returns the largest number which divides both of them) is associative, so

$$gcd(c, w_0, \ldots, w_{n-1}) = gcd(c, gcd(w_0, gcd(\ldots, gcd(w_{n-2}, w_{n-1}) \cdots)))$$

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*.

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

```
gcds :: (Int, Vector Int) -> Int
gcds (capacity, weights) = foldr gcd capacity weights
```

Fig. 3: Haskell

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`