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

July 1, 2014

Abstract This paper investigates the potential benefits offered by adopting a declarative approach, as embodied by modern functional languages with mature implementations, for prototyping algorithms for solving combinatorial optimisation problems. For this class of problems there are usually many different options for the core algorithms, supporting data structures and other implementation aspects. Thus tools that allow different alternatives to be tried out quickly, focusing on the essence of the problem, and as unencumbered as possible by implementation detail, would be very useful. As a case study, we consider dynamic programming algorithms. These have many uses in scheduling and timetabling: either directly, or as a component within other methods such as column generation. Our findings suggest that off-the-shelf, leading functional languages can indeed offer a range of compelling advantages in this particular problem domain, while yielding a performance that is adequate for verifying and evaluating the implemented algorithms as such.

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

1 Introduction

Over the last few decades, the speed of computers has increased by orders of magnitude, but the productivity of programmers has not kept pace. It is often far more important to quickly produce correct and robust code than to optimise code for performance. As computers continue to become more powerful this is ultimately going to become the norm. Prototyping new heuristics and algorithms for combinatorial optimisation is arguably one area where speed of development of correct code is already more important than absolute performance.

D. McGillicuddy, A. J. Parkes and H. Nilsson
School of Computer Science
University Of Nottingham
E-mail: {dxm, ajp, nhn}@cs.nott.ac.uk

We undertook a small case study as a preliminary investigation into whether a declarative approach, specifically functional programming, is feasible for this domain and whether it can help speed up prototyping. Our basic observation is that algorithms and heuristics for combinatorial optimisation at their core have clear mathematical specifications. Implementation, however, is often hampered by the need to spell out a plethora of operational details. This is time-consuming, error prone, and ultimately obscures the essence of the code. Thus if combinatorial optimisation algorithms could be prototyped by, for the most part, transliterating the mathematical specifications, and if the resulting performance were adequate for evaluation purposes, much would be gained already. Additionally because the elementary, "schoolbook" reasoning principle of substituting equals for equals is valid in declaratively formulated code, applying property-based testing (where test cases are derived automatically from stated correctness properties [1]), more easily exploiting multi-core architectures, and formally proving correctness, are potentially facilitated.

For our case study, we have opted to look at a few standard dynamic programming algorithms, specifically *unbounded knapsack* and *longest common substring* (LCS). These have many uses and, for our purposes, are representative of a larger class of algorithms in the domain of combinatorial optimisation. We have opted to use the pure, lazy, functional language Haskell as our declarative implementation framework [4]. Using a pure language increases the contrast to the imperative languages commonly used to implement this class of algorithms, making for a more interesting comparison, while also allowing the specific advantages of working declaratively to be fully realised. Further, Haskell is supported by mature, industrial-strength implementations, resulting in a fairer performance comparison [6].

We would like to emphasise that our aim is not to advocate any particular functional language for prototyping combinatorial optimisation algorithms. Rather, we are interested in exploring what advantages functional notation (supported by mature implementations) can bring today. However, it is worth noting that if these advantages are judged to be compelling enough, functional language implementations can, with relative ease, be leveraged for implementing domain-specific languages (DSL, sometimes referred to as 'executable specifications'). These allow domain-experts interested in working declaratively to reap the benefits of the approach without having to become seasoned functional programmers themselves [3]. One example of such a DSL, used to define and manipulate financial contracts, was produced by Simon Peyton Jones et al. and a derivative of it is used in industry by companies such as Bloomberg and HSBC Private Bank [5].

We carry out the study by implementing each of the chosen algorithms (unbounded knapsack and LCS) in Haskell, Java, and C. The implementations are then compared along a number of dimensions, including conciseness, modularity and performance, as well as ease of debugging, reasoning and parallelising. To make the comparisons meaningful we retain the structure the of the implementations across languages, except where we take advantage of specific language features (such as pointers, objects, or laziness). The implementations are further idiomatic and representative of what a "typical" programmer might write, without non-portable micro-optimisations. In particular, standard libraries are used throughout for data structures and mathematical computations, with as little as possible implemented from scratch.

2 An Illustrative Example

In a recent high profile case [2], a spreadsheet bug caused erroneous results from an economical analysis to be published, possibly influencing European Union policy[1]. The error was caused in part by an indexing mistake that accidentally excluded several countries from the analysis, a clear example of operational details causing problems. As an analogy, consider summing a collection of numbers. In a declarative setting the numbers (whether in the form of an array, list, stream, or otherwise) are simply passed to a generic *sum* function. Indexing and element-wise operations take place behind the scenes, completely eliminating these as possible sources of programmer error. By contrast, in most spreadsheets, the range of cells to be summed are manually selected (e.g., "C3:C100") which can be error-prone. Let us consider how similar ideas might improve a combinatorial optimisation algorithm. Lack of space precludes describing the full results of Knapsack and LCS, however, solving the unbounded knapsack problem involves finding the Greatest Common Divisor (gcd) of the initial capacity and an array $\mathbf{W}$ of n weights. The function *gcd*[2] is associative. Thus to get the gcd of the $n+1$ numbers, first the gcd of the capacity and W_0 is calculated, then the gcd of that number and W_1, and so on for each W_i, reducing to a single integer after n calls. Figure 1 shows the algorithm implemented in Java 7. Iteration over the elements has been abstracted into a *for-each* loop. The accumulator variable *gcd_all* is initialised to *capacity* and then *gcd*'d with each weight, updating the accumulator variable with the result of *gcd* for each $\mathbf{W}_i$. The C version of the algorithm is almost identical, except that the indices and loop ranges have to be written explicitly, adding further operational details. The Haskell version of *gcd* is shown in figure 2. Here the idiom of reducing a list by a binary function and accumulator is captured by the function `foldr1`, so called because it folds, associating to the right, over a list with at least one element. There is thus no need for the user to specify how and when the accumulator should be updated. Furthermore, since the definition of *gcd* contains the rule 'gcd 1 _ = 1', which states that $\forall x.gcd(1,x) = 1$, it can be said to be *short-circuiting*; i.e., if the first argument is equal to 1 then, due to lazy evaluation, the second argument is not inspected and is ignored. Therefore *gcds* will automatically stop once a 1 is encountered without any change to the loop itself. Achieving the same optimisation in Java (or C) would require changing the code for the loop itself by fusing it with part of the code *gcd*. This would break modularity, hamper reuse, and possibly render the code less readable. In this very small example, adding a check to see if *gcd_all* is equal to 1 at each iteration and halting the loop if so is a trivial change. However, had the loop or the called function been more involved, the modification would have been correspondingly harder because the code that governs the loop might be quite divorced from the code that updates the accumulating variable.

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

[2] Which takes two strictly positive integers and returns the largest integer that divides them both.

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

Fig. 1: Java 7

```
gcds :: Int -> [Int] -> Int
gcds capacity weights = foldr1 gcd (capacity:weights)
```

Fig. 2: Haskell

3 Results and Conclusions

Our findings so far, to be detailed in the full version of the paper, suggest that functional languages supported by mature implementations can indeed speed up development by allowing implementations to stay close to specifications, taking advantage of specific language features such as laziness, and eliminating certain classes of errors. Furthermore, they can achieve this without incurring a performance penalty that is unacceptable for prototypes. Our benchmark results for unbounded knapsack suggest that the C code is not more than about five times faster than the Haskell version. There are a wide range of languages that provide a transition path to more functional code; first-class functions, folds and pattern-matching have been added to object-oriented languages such as Java, C#, Scala and C++. F# and Clojure can interface seamlessly with C# and Java respectively, and both Haskell and Rust can easily interoperate with C. As such the authors recommend that readers familiarise themselves with these idioms and consider using them in their OR prototypes and implementations.

Acknowledgements This work was funded in part by EPSRC grant EP/F033613/1.

References

1. Claessen, K., Hughes, J.: QuickCheck: A lightweight tool for random testing of Haskell programs. SIGPLAN Not. **46**(4), 53–64 (2011). DOI 10.1145/1988042.1988046. URL `http://doi.acm.org/10.1145/1988042.1988046`
2. 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`
3. Hudak, P.: Modular domain specific languages and tools. In: Proceedings of Fifth International Conference on Software Reuse, pp. 134–142 (1998)
4. Jones, S.P. (ed.): Haskell 98 Language and Libraries – The Revised Report. Cambridge University Press, Cambridge, England (2003)
5. Jones, S.P., Eber, J.M., Seward, J.: Composing contracts: an adventure in financial engineering (functional pearl). ACM SIGPLAN NOTICES **35**(9), 280–292 (2000)
6. Terei, D.A., Chakravarty, M.M.: An llvm backend for ghc. In: ACM Sigplan Notices, vol. 45, pp. 109–120. ACM (2010)