Submitted to IFL '13

Structured Sharing for Dynamic Programming

Extended Abstract

Nicolas Wu *

Department of Computer Science
University of Oxford
nicolas.wu@cs.ox.ac.uk

Abstract

Dynamic programming is a technique for designing algorithms to solve certain optimisation problems. It is similar to the divide-and-conquer method in that a problem is split into subproblems that are recursively solved, and the subsolutions are combined to form a final solution. However, unlike divide-and-conquer, the subproblems are not necessarily independent. Indeed, the efficiency of dynamic programming algorithms relies on sharing the solutions to subproblems that are revisited. The usual approach to dynamic programming is to use an array to store the subsolutions. The construction of the array begins with base cases, and must carefully respect the dependencies that subsolutions have.

As a motivating example, we will be considering the *minimum edit distance* problem, which nicely demonstrates how a recursive definition can be turned into a more efficient dynamic algorithm. In this problem we are concerned with finding the minimal edit distance of two strings. One such measure is the Levenshtein distance, which is the minimum number of substitutions, insertions, and deletions of single characters that can be made to turn one string into another.

$$
\begin{aligned}
&\mathit{editDist}_1 :: (\mathit{String}, \mathit{String}) \to \mathbb{N}\\
&\mathit{editDist}_1\ (xs, [\,]) = \mathit{length}\ xs\\
&\mathit{editDist}_1\ ([\,], ys) = \mathit{length}\ ys\\
&\mathit{editDist}_1\ (xs@(x : xs'), ys@(y : ys')) =\\
&\quad \mathit{minimum}\ [\mathit{editDist}_1\ (xs,\ ys') + 1, \mathit{editDist}_1\ (xs', ys) + 1,\\
&\qquad\qquad \mathit{editDist}_1\ (xs', ys') + \mathbf{if}\ x == y\ \mathbf{then}\ 0\ \mathbf{else}\ 1]
\end{aligned}
$$

In the base cases, a string can be turned into an empty one by deleting all its characters (or conversely, an empty string can be turned into a string by inserting that those characters), which takes as many operations as there are characters. When both strings are non-empty we have choices: we can delete the first character from either *xs* or *ys* or we can substitute the first character of *xs* for that of *ys* if they differ. Whatever the choice, we add the cost of the operation to the minimum distance of the ensuing strings.

Such simple recursive equations are often a good starting point when describing algorithms, but can be terribly inefficient. In this case, we can save ourselves exponential work by sharing the results of previously encountered subproblems.

The typical means of improving efficiency is to tabulate the intermediate results in an array that is indexed by the parameters of the function. Here, however, the parameter is a pair of strings, and indexing such values in an array is not generally desirable. We can massage this problem away by instead passing indices that indicate the remaining length of the string that is being inspected.

$$
\begin{aligned}
&\mathit{editDist}_2 :: (\mathit{String}, \mathit{String}) \to \mathbb{N}\\
&\mathit{editDist}_2\ (xs, ys) = \mathit{table}\,!\,mn\ \mathbf{where}\\
&\quad mn = (\mathit{length}\ xs, \mathit{length}\ ys)\\
&\quad \mathit{table} = \mathit{tabulate}\ ((0,0), mn)\ \mathit{editDist}\\
&\quad \mathit{editDist}\ (i, 0) = i\\
&\quad \mathit{editDist}\ (0, j) = j\\
&\quad \mathit{editDist}\ (i@(i'+1), j@(j'+1)) =\\
&\qquad \mathit{minimum}\ [\mathit{table}\,!\,(i, j')\ + 1, \mathit{table}\,!\,(i', j) + 1,\\
&\qquad\qquad \mathit{table}\,!\,(i', j') + \mathbf{if}\ xs\,!!\,i' == ys\,!!\,j'\ \mathbf{then}\ 0\ \mathbf{else}\ 1]
\end{aligned}
$$

This makes use of the function *tabulate*, which constructs a table that is mutually dependant on the function *editDist*.

$$
\begin{aligned}
&\mathit{tabulate} :: (\mathit{Ix}\ i) \Rightarrow (i, i) \to (i \to e) \to \mathsf{Array}\ i\ e\\
&\mathit{tabulate}\ ixs\ f = \mathit{array}\ ixs\ [(i, f\,i) \mid i \leftarrow \mathit{range}\ ixs]
\end{aligned}
$$

Although this code mirrors its specification quite closely, we shall see that by moving to a more abstract setting, we can do better yet.

Our general approach will be to use structured recursion schemes. The seed for many of these ideas finds its roots in histomorphisms, introduced as a recursion scheme for course-of-values recursion by Uustalu and Vene [8]. For dynamic programming, however, this scheme suffers from a serious limitation: the coalgebra is restricted to be the inverse of an initial algebra, which rules out its application to many problems. The restriction on the coalgebras was later lifted by Kabanov and Vene [7] with the introduction of dynamorphisms, although the development was restricted to the setting of CPO. However, it was known at the time that histomorphisms, which are closely related, are an instance of recursion schemes from comonads [9], which in turn can be generalised to the setting of recursive coalgebras [3]. A somewhat surprising result by Hinze et al. [6] showed that recursive schemes from comonads could be unified with adjoint folds. This work was applied directly to dynamic programming by Hinze and Wu [4], where bialgebras and dicoalgebras played a central role of the proofs. The setting of adjoint folds was generalized to cover all forms of conjugate hylomorphisms by Hinze et al. [5], where further examples of dynamic programming were given. In particular, they showed how algorithms with finite lookup tables can be expressed in terms of para-hylos. There remains, however, one wrinkle in the story: in all these approaches, the intermediate values are stored in structures with linear lookup times.

* This work has been funded by EPSRC grant number EP/J010995/1.

The aim of this work is to show how structured recursion schemes can be used to express dynamic programming algorithms without resorting to linear lookup structures. We do so by building an intermediate structure with sharing. The result can be thought of as a nexus, which were discussed in a similar context by Bird and Hinze [2]. Bird [1] later showed how nexuses could be used as the intermediate structure of hylomorphisms.

For a taste of the development to come, we will be working with the following base functor that more closely follows the structure of the recursion:

$$\begin{aligned}&\textbf{data}\ \mathsf{EditDist}\ x = \mathsf{Base}\ \mathit{String} \mid \mathsf{Other}\ \mathit{Char}\ \mathit{Char}\ x\ x\ x\\&\textbf{instance}\ \mathit{Functor}\ \mathsf{EditDist}\ \textbf{where}\\&\quad \mathit{fmap}\ f\ (\mathsf{Base}\ xs) = \mathsf{Base}\ xs\\&\quad \mathit{fmap}\ f\ (\mathsf{Other}\ x\ y\ u\ v\ w) = \mathsf{Other}\ x\ y\ (f\,u)\ (f\,v)\ (f\,w)\end{aligned}$$

The following algebra for this base functor performs the work of turning problems into solutions.

$$\begin{aligned}&\mathsf{editDist_A} :: \mathsf{EditDist}\ \mathbb{N} \to \mathbb{N}\\&\mathsf{editDist_A}\ (\mathsf{Base}\ xs) = \mathit{length}\ xs\\&\mathsf{editDist_A}\ (\mathsf{Other}\ x\ y\ u\ v\ w) =\\&\quad \mathit{minimum}\ [u+1, v+1, w+\textbf{if}\ x == y\ \textbf{then}\ 0\ \textbf{else}\ 1]\end{aligned}$$

This algebra can be used to deconstruct values that have been set up by the corresponding coalgebra:

$$\begin{aligned}&\mathsf{editDist_C} :: (\mathit{String}, \mathit{String}) \to \mathsf{EditDist}\ (\mathit{String}, \mathit{String})\\&\mathsf{editDist_C}\ (xs, [\,]) = \mathsf{Base}\ xs\\&\mathsf{editDist_C}\ ([\,], ys) = \mathsf{Base}\ ys\\&\mathsf{editDist_C}\ (xs@(x : xs'), ys@(y : ys')) =\\&\quad \mathsf{Other}\ x\ y\ (xs, ys')\ (xs', ys)\ (xs', ys')\end{aligned}$$

Now, it so happens that in Haskell we can use both the fold of the algebra and the unfold of the coalgebra to obtain the following (inefficient) solution:

$$\begin{aligned}&\mathit{editDist_3} :: (\mathit{String}, \mathit{String}) \to \mathbb{N}\\&\mathit{editDist_3} = (\!|\mathsf{editDist_A}|\!) \cdot [\!|\mathsf{editDist_C}|\!]\ .\end{aligned}$$

We will show how the intermediate datastructure can be generated from the cofree comonad, rather than directly from the unfolding of the coalgebra $\mathsf{editDist_C}$, and further demonstrate how this can be done in such a way that sharing is preserved between nodes. Much of this work is inspired by categorical machinery that may not be familiar to most programmers. Our goal is to keep the use of such machinery light, and to focus on the details of practical implementation, rather than on the body of theory that is already established.

References

[1] R. Bird, *Pearls of Functional Algorithm Design*. Cambridge University Press, 2010.

[2] R. Bird and R. Hinze, "Functional pearl: Trouble shared is trouble halved," in *Haskell '03*. New York, NY, USA: ACM, 2003, pp. 1–6. doi:10.1145/871895.871896

[3] V. Capretta, T. Uustalu, and V. Vene, "Recursive coalgebras from comonads," *Information and Computation*, vol. 204, no. 4, pp. 437–468, 2006. doi:10.1016/j.ic.2005.08.005

[4] R. Hinze and N. Wu, "Histo- and dynamorphisms revisited," in *Workshop on Generic Program.* ACM, Sep. 2013. doi:10.1145/2502488.2502496

[5] R. Hinze, N. Wu, and J. Gibbons, "Conjugate hylomorphisms," Jul. 2013, in submission.

[6] ——, "Unifying structured recursion schemes," in *International Conference on Functional Programming*. ACM, Sep. 2013. doi:10.1145/2500365.2500578

[7] J. Kabanov and V. Vene, "Recursion schemes for dynamic programming," in *Mathematics of Program Construction, 8th International Conference, MPC 2006*. Springer, 2006, pp. 235–252. doi:10.1007/11783596_15

[8] T. Uustalu and V. Vene, "Primitive (co)recursion and course-of-value (co)iteration, categorically," *Informatica, Lith. Acad. Sci.*, vol. 10, no. 1, pp. 5–26, 1999.

[9] T. Uustalu, V. Vene, and A. Pardo, "Recursion schemes from comonads," *Nordic J. of Computing*, vol. 8, no. 3, pp. 366–390, Sep. 2001.