Received: Added at production | Revised: Added at production | Accepted: Added at production

https://doi.org/10.1002/spe.2887

RESEARCH ARTICLE

Easily Solving Dynamic Programming Problems in Haskell by Memoization of Hylomorphisms

David Llorens | Juan Miguel Vilar*

Institute of New Image Technologies,
Universitat Jaume I, SPAIN

Correspondence
Juan Miguel Vilar
Institute of New Image Technologies
Universitat Jaume I
12071, Castelló de la Plana (SPAIN)
Email: jvilar@uji.es

Dynamic Programming is a well known algorithmic technique that solves problems by a combination of dividing a problem into subproblems and using memoization to avoid an exponential growth of the costs. We show how to implement Dynamic Programming in Haskell using a variation of hylomorphisms that includes memoization. Our implementation uses polymorphism so the same function can return the best score or the solution to the problem based on the type of the returned value.

KEYWORDS:
Dynamic Programming, recursion schemes, Haskell, memoization

1 | INTRODUCTION

It is very likely that the reader knows about *Dynamic Programming*, which is explained in many introductory books on algorithms like Cormen et al.[1] or Rabhi and Lapalme[2]. It is used in many contexts since it succeeds in reducing the time costs of several algorithms from exponential to polynomial (or pseudopolynomial) incurring in small spatial costs. Two well known examples are the knapsack and the edit distance problems.

The reader is probably also aware that the reason behind this dramatic cost reduction is the use of memoization for avoiding the repetition of computations in problems where the solution to an instance can be found by the combination of the solution to smaller instances. For instance, as explained later, the best way to fill a knapsack can be found by considering the last item and deciding whether to add it to the items that optimally fill a smaller knapsack or not to add the last item to the best solution for the original knapsack with the rest of the items.

However, it may be challenging to find the commonalities of several such problems so a single library can solve them. Just consider how would the reader use the same function to solve the mentioned knapsack and edit distance problems.

The purpose of this paper is twofold: we show that the use of semirings allows the expression of the commonalities for a lot of different Dynamic Programming problems and use that formalism together with the notion of hylomorphism to write a Haskell library that solves those problems. This library has in turn two main strengths: first, it only needs the description of the problem in terms of how an instance of a problem is divided into smaller ones and how the solutions to those instances is combined to recover the solution to the original problem; second, the use of different semirings, implemented as different types, allows the user to obtain different results like the best total value of the items that fit in the knapsack, the list of such items, or the edit operations needed to transform one string into other.

The use of Haskell allows us to implement these concepts in a very concise way. The reader is only assumed to have an intermediate level of Haskell knowledge to fully understand the details of the implementation and an introductory level to use the library. Familiarity with type classes, including the most common ones, specially the `Functor` class, is expected and it is the most advanced concept from Haskell used. We use several extensions present in the most popular Haskell compiler, GHC[3], but they are introduced when needed.

2 | SOME PROBLEMS

Let us begin our exposition by studying three classical Dynamic Programming problems so that we can later generalize their solutions.

2.1 | Knapsack

As a first example for the presentation of our approach, we use the discrete knapsack problem. We assume that we have a knapsack with a given capacity and a list of items each having a value and a weight. In Haskell we declare the following types:

```
type Capacity = Int
type Value = Int
type Weight = Int
data Item = Item { value :: Value, weight :: Weight }
data KSProblem = KSP { capacity :: Capacity, items :: [Item] }
```

To find the subset of items that fits in the knapsack and has the highest total value, we can consider two options: either we drop the first item and find the best subset of the rest of the items for the original knapsack or we pick the first item (if it will fit) and find the best subset of the rest for a knapsack with smaller capacity. Then, we keep the option with the best value. If we only want to know the highest value, we just keep the maximum of the two values. This can easily be expressed as

```
knapsack :: KSProblem -> Value
knapsack (KSP c []) = 0
knapsack (KSP c (i:is)) | weight i <= c = max (knapsack (KSP c is))
                                              (value i + knapsack (KSP (c-weight i) is))
                        | otherwise = knapsack (KSP c is)
```

2.2 | Edit Distance

Now, let us introduce a second example, the computation of the edit distance between two strings s and t. The problem is to find the least number of edit operations needed to transform s into t, where the available edit operations over the current string are:

Addition of a symbol.

Deletion of a symbol.

Substitution of a symbol for another.

For instance, the transformation of “train” into “basin” can be accomplished with just three operations:

- Substitution of the “t” by the “b”: “train” → “brain”.
- Deletion of the “r”: “brain” → “bain”.
- Insertion of an “s”: “bain” → “basin”.

A way of finding this sequence is to consider the three options for the first position of the string and derive from each one a new problem that can be solved recursively. In our case the options are:

- Substitute “t” by “b” and add one to the edit distance between “rain” and “asin”.
- Delete the “t” and add one to the edit distance between “rain” and “basin”.

- Insert a “b” and add one to the edit distance between “train” and “asin”

Let us define the following types:

```
type EDProblem = (String, String)
type Distance = Int
```

Then, we can define the function `editDistance` as:

```
editDistance :: EDProblem -> Distance
editDistance ([], []) = 0
editDistance (x:xs, []) = 1 + editDistance (xs, [])
editDistance ([], y: ys) = 1 + editDistance ([], ys)
editDistance (x:xs, y:ys) = minimum [ s + editDistance (xs, ys)
                                    , 1 + editDistance (x:xs, ys)
                                    , 1 + editDistance (xs, y:ys) ]
                            where s = if x == y then 0 else 1
```

We use `s` to indicate whether the initial characters of the string are equal (and therefore, there is no operation and no cost) or different and the operation has a unit cost.

2.3 | Random Walk

The last problem for this section will be a random walk on the integer number line. We have an object that starts at a given position p and can move forward or backward with equal probability. We want to know the probability that the object reaches a position $f \geq p$ in s steps or fewer. So we define the types[1]:

```
type Position = Int
type Step = Int
newtype Probability = Probability Double deriving (Show, Eq, Ord, Fractional, Num)
data RWProblem = RW { from :: Position, to :: Position, remaining :: Step }
```

When the item has reached the final position, the probability that it happens is one. On the other hand, if it is not the case and there are no steps left, the probability is zero. In the general case, we have to consider two possibilities: either the object moves forward and then we are asking the same question but departing from $p + 1$; or it moves backward and the new initial position is $p - 1$. Each possibility happens with probability one half, so we can write the function for computing the probability as

```
randomWalk :: RWProblem -> Probability
randomWalk (RW p f s) | p >= f = 1
                      | s == 0 = 0
                      | otherwise = sum [ 0.5 * randomWalk (RW (p + 1) f (s - 1))
                                        , 0.5 * randomWalk (RW (p - 1) f (s - 1))
                                        ]
```

3 | UNIFYING THE PROBLEMS

All the previous problems have a common structure in that they take an instance of a problem and to solve it, first they check if it is “small enough” and give a trivial answer. If it is not, they consider a small set of decisions (to take or drop an item, which edit operation to apply, whether to go forward or backward) and for each decision, they have to combine the score of that decision with the score of the instance that results of making that decision. We can make the commonality clearer by rewriting the functions like this:

[1] To be able to automatically derive the `Num` and `Fractional` instances, the language pragma `GeneralizedNewtypeDeriving` is used.

```
knapsack :: KSProblem -> Value
knapsack p
  | isTrivial p = 0
  | otherwise = maximum [ s + knapsack sp | (s, sp) <- subproblems p ]
  where
    isTrivial = null . items
    subproblems (KSP c (i:is))
      | value i <= c = [(0, KSP c is), (value i, (KSP (c-weight i) is))]
      | otherwise = [(0, KSP c is)]

editDistance :: EDProblem -> Distance
editDistance p
  | isTrivial p = 0
  | otherwise = minimum [ w + editDistance sp | (w, sp) <- subproblems p ]
  where isTrivial = ( == ([], []))
        subproblems (x:xs, []) = [ (1, (xs, [])) ]
        subproblems ([], y:ys) = [ (1, ([], ys)) ]
        subproblems (x:xs, y:ys) = [ (s, (xs, ys))
                                   , (1, (x:xs, ys))
                                   , (1, (xs, y:ys))]
          where s = if x == y then 0 else 1

randomWalk :: RWProblem -> Probability
randomWalk p
  | isTrivial p = 1
  | otherwise = sum [ pr * randomWalk sp | (pr, sp) <- subproblems p ]
  where isTrivial (RW p f _) = p >= f
        subproblems (RW _ _ 0) = []
        subproblems (RW p f s) = [ (0.5, RW (p + 1) f (s - 1))
                                 , (0.5, RW (p - 1) f (s - 1))
                                 ]
```

Note that in `randomWalk` we have used the fact that the sum of the empty list is zero.

4 | SEMIRINGS

It is clear that all three functions are very similar but we need a way of unifying the expressions used both in the combination of the decisions with the results (addition for `knapsack` and `editDistance`, product for `randomWalk`) of the smaller subproblems and the combination of all the results (maximum for `knapsack`, minimum for `editDistance`, and sum for `randomWalk`). Also, we need to express the result of a trivial instance. The algebraic structure to accomplish this is the semiring.

A *semiring* is a set S with two operations, $\oplus$ and $\otimes$ and two distinguished elements, 0 and 1, such that the following hold for all $a, b, c \in S$:

- S with $\oplus$ is a commutative monoid with 0 as the identity, i.e. $\oplus$ is commutative, $a \oplus b = b \oplus a$; associative, $(a \oplus b) \oplus c = a \oplus (b \oplus c)$; and $a \oplus 0 = 0 \oplus a = a$.
- S with $\otimes$ is a monoid with 1 as the identity, i.e. $\otimes$ is associative, $(a \otimes b) \otimes c = a \otimes (b \otimes c)$; and $a \otimes 1 = 1 \otimes a = a$.
- The operation $\otimes$ distributes over $\oplus$, $a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$.
- The element 0 is absorbent for $\otimes$, $a \otimes 0 = 0 \otimes a = 0$.

The operation $\oplus$ corresponds to the combination of the results. Its commutative and associative properties allow the generation of the results in any order without altering the results, so for instance, it does not matter in which order the different knapsacks are generated. The identity element for $\oplus$ allows the scoring of the situation in which no solution is available. For instance, if there is no previous knapsack and one is produced, the best option is to take the new knapsack.

Similarly, the associative property for $\otimes$, that represents the combination of the different decisions allows an implementation to go in either direction, i.e. we can consider the value of a knapsack either as the value of the first element plus the value of the rest of the elements or as the value of all the elements except for the last plus the value of that last element. The identity element $\otimes$ allows starting the scoring process by an empty solution. For instance, the value of adding an item to an empty knapsack is just the value of that item.

In Haskell, we define the class `Semiring` like this:

```
class Semiring s where
    infixl 6 <+>
    (<+>) :: s -> s -> s
    infixl 7 <.>
    (<.>) :: s -> s -> s
    zero  :: s
    one   :: s
```

The instance for `Probability` is direct:

```
instance Semiring Probability where
  (<+>) = (+)
  (<.>) = (*)
  zero = 0
  one = 1
```

However, for `Value` and `Distance` we have a small conflict, since the combination of two `Values` returns their maximum and the combination of two `Distances` returns their minimum. This situation arises often. For instance, the integers with addition form a monoid while with product form a different one. The solution given in the `Data.Monoid` library is to use `newtype` to wrap the values in `Sum` for addition and `Product` for product. In our case, we have two examples of the so called *tropical semirings*; so we define two generic new types to signal whether the intention is to maximize or minimize and to allow also working with types different from `Int`.

```
newtype TMin v = TMin v deriving (Eq, Ord, Show)
newtype TMax v = TMax v deriving (Eq, Ord, Show)
```

The corresponding instances are

```
instance (Num v, Ord v, Bounded v) => Semiring (TMin v) where
  t1 <+> t2 = min t1 t2
  t1@(TMin v1) <.> t2@(TMin v2)
    | t1 == zero = zero
    | t2 == zero = zero
    | otherwise = TMin (v1 + v2)
  zero = TMin maxBound
  one = TMin 0

instance (Num v, Ord v, Bounded v) => Semiring (TMax v) where
  t1 <+> t2 = max t1 t2
  t1@(TMax v1) <.> t2@(TMax v2)
    | t1 == zero = zero
    | t2 == zero = zero
    | otherwise = TMax (v1 + v2)
  zero = TMax minBound
  one = TMax 0
```

Here the definition of `<.>` is slightly complicated by the fact that we are using *saturating addition* (in case you are worried, in the uses we will give to them, the associate property will not be violated.)

Now we can declare a Dynamic Programming Problem as a record that contains the initial instance of the problem and the two functions `isTrivial` and `subproblems`:

```
data DPProblem p sc = DPProblem { initial     :: p
                                , isTrivial   :: p -> Bool
                                , subproblems :: p -> [(sc, p)]
                                }
```

Using this record, the general solution to a `DPProblem` is given by this function:

```
dpSolve :: Semiring sc => DPProblem p sc -> sc
dpSolve dp = go $ initial dp
  where go p | isTrivial dp p = one
             | otherwise = foldl' (<+>) zero [ sc <.> go sp
                                             | (sc, sp) <- subproblems dp p ]
```

We can now reuse our previous implementations of `isTrivial` and `subproblems` to define the corresponding `DPProblem` and rewrite the `knapsack` function like this:

```
ksDPProblem :: KSProblem -> DPProblem KSProblem (TMax Value)
ksDPProblem p = DPProblem p isTrivial subproblems
  where
    isTrivial = null . items
    subproblems (KSP c (i:is))
      | value i <= c = [(TMax 0, KSP c is), (TMax (value i), KSP (c-weight i) is)]
      | otherwise = [(TMax 0, KSP c is)]

knapsack :: KSProblem -> TMax Value
knapsack = dpSolve . ksDPProblem
```

The rewriting of the functions for the edit distance problem and the random walk are also straightforward, but in this case we have to introduce the new types `TMin` and `TMax`:

```
edDPProblem :: EDProblem -> DPProblem EDProblem (TMin Distance)
edDPProblem p = DPProblem p isTrivial subproblems
  where isTrivial = ( == ([], []))
        subproblems (x:xs, []) = [(TMin 1, (xs, []))]
        subproblems ([], y:ys) = [(TMin 1, ([], ys))]
        subproblems (x:xs, y:ys) = [ (s, (xs, ys))
                                   , (TMin 1, (x:xs, ys))
                                   , (TMin 1, (xs, y:ys))]
          where s = TMin $ if x == y then 0 else 1

editDistance :: EDProblem -> TMin Distance
editDistance = dpSolve . edDPProblem

rwDPProblem :: RWProblem -> DPProblem RWProblem Probability
rwDPProblem p = DPProblem p isTrivial subproblems
  where isTrivial p = from p >= to p
        subproblems (RW _ _ 0) = []
        subproblems (RW p f s) = [ (0.5, RW (p + 1) f (s - 1))
                                 , (0.5, RW (p - 1) f (s - 1))
```

```
                                       ]

randomWalk :: RWProblem -> Probability
randomWalk = dpSolve . rwDPProblem
```

5 | HYLOMORPHISMS

Now let us concentrate on the `dpSolve` function. This is a function that can be easily written as a *hylomorphism*, see Meijer et al.[4]. This can be considered as an abstraction of the divide-and-conquer structure of computation.

To explain the concept of hylomorphism, we can imagine that we have a function `p` → `s` that can be computed by first expanding the `p` into several more values of type `p`, then computing recursively the function on those values, and finally combining the results so obtained into the final result. For instance, computing the value of the Fibonacci function in n can be accomplished by expanding n into $n-1$ and $n-2$, then computing the function on them, and finally adding those values to get the final result.

To capture the structure of those functions, we need a way to express three different concepts: the process of producing several values, the application of the function to those values, and the combination of the result of the application. The reader is probably aware that the class `Functor` captures nicely the idea of a structure for storing values and applying a function to each one of them.

For the process of producing a single value from a functor we can use the concept of *algebra* (or more formally, *f-algebra*), a function from a functor to its carrier type. In Haskell:

```
type Algebra f s = f s -> s
```

Without going into much detail, the name comes from the fact that algebras abstract algebraic structures. For instance, consider the monoid `Int` with operation `(+)` and identity `1`. It can be represented by the function `m :: Either (Int, Int) () -> Int`, where `m (Left a b) = a+b` and `m (Right ()) = 1`. They are also used in computing for representing data structures, like lists, which are initial algebras of certain functor. However, in order to understand how we will use them, we can say that they represent the process of computing a value from a collection.

Similarly, as producing a collection of values from a single one is dual to producing a value from a collection, we can use the dual concept of algebra, the *coalgebra*, which is a function from a type to a functor. In Haskell:

```
type Coalgebra f p = p -> f p
```

Although not as popular as algebras, there are many aspects of computing that are modeled by coalgebras like modeling infinite structures like streams, the transition between states, or objects (in the OOP sense). For instance, consider an object `C` with a single integer attribute `i`. Then the setter and getter for `i` are functions `C -> Int` and `C -> Int -> C`, and they can be unified in a function `C -> (Int, Int -> C)`. Again, like in the case of algebras, we only need the simplified interpretation of the coalgebra as the producer of a collection of values from a single one.

Returning to hylomorphisms, they take an algebra on solutions and a coalgebra on problems, and construct a function from problems to solutions. This function solves a problem p in three steps: first, it uses the coalgebra to obtain from p a collection of new problems; second, the collection of the solutions to these problems is found by recursively repeating the process over the collection using `fmap`; third, the final solution to p is obtained by using the algebra on the collection of solutions.

In Haskell:

```
hylo :: Functor f => Algebra f s -> Coalgebra f p -> p -> s
hylo alg coalg = h
     where h = alg . fmap h . coalg
```

For example, consider the Fibonacci function. Given n there are two possible situations: if $n \le 2$ we have reached the base case; otherwise, the final result can be computed using the value of the function in $n-1$ and $n-2$. We can express these two possibilities with the following type:

```
data Fib a = Base | Pair a a
```

```
instance Functor Fib where
  fmap _ Base = Base
  fmap f (Pair a b) = Pair (f a) (f b)
```

And the corresponding coalgebra takes *n* and either returns `Base` or the `Pair (n-1) (n-2)`. Similarly, the algebra for `Base` returns 1 and `a+b` for a `Pair a b`. Putting it all together we arrive at:

```
fibonacci :: Int -> Integer
fibonacci = hylo alg coalg
  where alg Base = 1
        alg (Pair a b) = a + b
        coalg n | n <= 2 = Base
                | otherwise = Pair (n-1) (n-2)
```

Note that the parameter of the function is `Int`, corresponding to limited range integers, and the output is `Integer`, which is used to represent unlimited integers. The use of these two different types is appropriate for the expected values of the input and output of the function but it is also interesting because it allows us to show that the functor `Fib` is applied to two different types: the value returned from the coalgebra is a `Fib Int` as it correspond to the input values; the value accepted by the algebra is a `Fib Integer`. And the transformation from one type to the other happens in the application of `fmap` within `hylo`.

Now we can rewrite `dpSolve` using `hylo`. We need a functor to store first the subproblems and then their solutions. As we need to remember the scores of the decisions corresponding to those solutions, we use a list of pairs:

```
data DPF sc p = Trivial | Children [(sc, p)]

instance Functor (DPF sc) where
  fmap _ Trivial = Trivial
  fmap f (Children cs) = Children [(sc, f p) | (sc, p) <- cs]
```

`Trivial` corresponds to the trivial problems whereas `Children` is used to store the children together with the scores associated to them. It may seem that `Trivial` is not needed if it were codified as `Children []`, but this is not possible since the meaning of an empty list in `Children` is that the corresponding problem is not feasible. We have seen an example of this in the case of a random walk with zero steps left when the final position has not been reached. Another example of this possibility is the problem of text segmentation discussed in Section 9.2.

The algebra, which we will call `solve`, will receive either a `Trivial` value, in which case it must return `one`, or a list of `Children` that must be combined. This is accomplished in two steps: the operation `<.>` has to be applied to each of the pairs (score, solution). Then all the values are joined together using `(<+>)`, which is accomplished by using a *fold*, in our case `foldl'`, which can be understood as taking the first two elements, adding them, then adding the third one and so on until the list is exhausted.

The definition of `solve` is then:

```
solve Trivial = one
solve (Children sols) = foldl' (<+>) zero [ sc <.> sol | (sc, sol) <- sols ]
```

The coalgebra, called `build`, is responsible for returning `Trivial` if the instance is small enough (i.e. if `isTrivial` returns `True`) or the list returned by `subproblems` for bigger instances. This can be easily written as:

```
build p | isTrivial dp p = Trivial
        | otherwise = Children $ subproblems dp p
```

We can use both definitions to rewrite `dpSolve` as:

```
dpSolve :: Semiring sc => DPProblem p sc -> sc
dpSolve dp = hylo solve build $ initial dp
  where build p | isTrivial dp p = Trivial
```

```
                | otherwise = Children $ subproblems dp p
        solve Trivial = one
        solve (Children sols) =
           foldl' (<+>) zero [ sc <.> sol | (sc, sol) <- sols ]
```

6 | MEMOIZATION

The defining characteristic of Dynamic Programming is the use of memoization to reduce the temporal costs from (typically) exponential to (hopefully) polynomial.

One advantage of Haskell is that the introduction of memoization can be done easily, as demonstrated in Hinze[5]. Furthermore, the ideas from that paper are implemented in the package `Data.MemoTrie`[6]. The `memo` function has type

```
HasTrie t => (t -> a) -> t -> a
```

so memoizing `f` is a simple as writing `memo f`. As it can be gathered from the type, the result of `memo f` is another function with the same type of `f`. The difference is that this new function will store the results of any call to `f` so that the computations are performed only once per input value.

The `HasTrie` class has instances for the most common types and can be easily derived for algebraic types as shown in the Appendix A.

In our case, we need to insert that memoization within `hylo`. This is accomplished by the following definition of `hyloM` (hylo with Memoization):

```
hyloM :: (Functor f, HasTrie s) => Algebra f t -> Coalgebra f s -> s -> t
hyloM alg coalg = h
     where h = memo $ alg . fmap h . coalg
```

This memoizes `h`, which is the real "core" of `hyloM`.

The improvement of speed of `hyloM` with respect to `hylo` can be dramatic as exemplified in the following extract of a session of GHCi (an interpreter for Haskell):

```
*Main> :set +s
*Main> fibonacci 40
102334155
(141.27 secs, 103,153,224,824 bytes)
*Main> fibonacciM 40
102334155
(0.01 secs, 464,072 bytes)
*Main>
```

The first line makes GHCi display statistics about the execution of each expression. The `fibonacci 40` uses the version of fibonacci shown in Section 5. The second call (`fibonacciM 40`) is the same function but with hylo replaced by `hyloM`.

Now, to use this new version of `hylo` we only need to change a line in `dpSolve`:

```
...
dpSolve dp = hyloM solve build $ initial dp
...
```

7 | RECOVERING THE SOLUTIONS

Usually, we are interested not only in the score of the best solution but in the solution itself. For instance, in the knapsack problem we want to know which items to pick. A possible approach will be to change the signature of `dpSolve` so that it returns a pair with the score and the sequence of decisions taken. However, this approach does not work for problems in which there is no "solution", like in the case of the random walk. The alternative we will use is to include the desired information within the semiring returned by `dpSolve`.

Consider the knapsack and the edit distance problems. In both cases, the desired outcome is a pair with the sequence of decisions and the corresponding score. In the knapsack problem, we can represent a decision with `Just i` if the item `i` was taken and with `Nothing` if not. In edit distance, we can define a type that represents the decisions as operations:

```
data EDOperation = Ins Char | Del Char | Replace Char Char | Keep Char
```

The first change we have to make is to include the type of the decisions in `DPProblem` and change `subproblems` so that each subproblem contains the decision associated with it.

```
data DPProblem p sc d = DPProblem { initial :: p
                                  , isTrivial :: p -> Bool
                                  , subproblems :: p -> [(sc, d, p)]
                                  }
```

Keeping track of the decisions allows us to recover the solution without changing the definition of the problem. As before, we use `Value` for the type of the scores and decide later whether we want to maximize or minimize it.

```
ksDPProblem :: KSProblem -> DPProblem KSProblem Value (Maybe Item)
ksDPProblem p = DPProblem p isTrivial subproblems
  where
    isTrivial = null . items
    subproblems (KSP c (i:is))
      | value i <= c = [(0, Nothing, KSP c is) , (value i, Just i, (KSP (c-weight i) is))]
      | otherwise = [(0, Nothing, KSP c is)]
```

As mentioned above, `Just i` means that the item `i` is included in the knapsack and `Nothing` that it is not.

To relate the score of the decisions to the solution, we define a class with a single function that combines the score of the decision with the decision itself to produce the part of the solution containing that decision:

```
class DPTypes sc d sol where
    combine :: sc -> d -> sol
```

The idea is that the same score for a decision can produce solutions of different types and `combine` associates decisions and solutions. For instance, the score of a decision can be an `Int`, but the solution for a maximization problem will be a `TMax Int` while for a minimization problem the solution will be a `TMin Int`. And there is also the case in which we want the solution to include the decisions made, which we will handle in a moment with the introduction of a new semiring. Furthermore, `DPTypes` will make it possible for `dpSolve` to return results of different types as needed.

The class `DPTypes` uses several advanced features that must be introduced in the language by using the corresponding language pragmas. In paticular, we use `MultiParamTypeClasses` to allow having more than one type parameter and `FlexibleInstances` and `FlexibleContexts` to be able to define the instances.

When we are only interested in the final score, we can choose the type of the solution to be the same as that of the scores, while the decision can simply be ignored:

```
instance DPTypes sc d sc where
    combine = const
```

Similarly, the instances for maximizing or minimizing also ignore the decision but tag the score with the appropriate constructor:

```
instance DPTypes sc d (TMin sc) where
    combine = const . TMin

instance DPTypes sc d (TMax sc) where
    combine = const . TMax
```

The final version of dpSolve is:

```
dpSolve :: (HasTrie p, Semiring sol, DPTypes sc d sol)
             => DPProblem p sc d -> sol
dpSolve dp = hyloM solve build $ initial dp
  where build p | isTrivial dp p = Trivial
                | otherwise = Children [(combine sc d, sp) |
                                        (sc, d, sp) <- subproblems dp p]
        solve Trivial = one
        solve (Children sols) =
           foldl' (<+>) zero [ sc <.> sol | (sc, sol) <- sols ]
```

The differences with the previous version are that the return type is different from that of the scores and that, due to this, the build function now combines the scores with the decisions.

To recover the list of items in the knapsack, we can now create a new semiring. First, we define a data type to hold the list of decisions and its score:

```
data BestSolution d sc = BestSolution (Maybe [d]) sc deriving Show
```

Note that we use Maybe because the problem may have no solution (which is different from having the empty list as solution). As we already mentioned, an example of that possibility is the problem of text segmentation discussed in Section 9.2.

The plus operation for BestSolution must choose the solution with the best score from its two arguments, therefore we need to define a type class that reflects the concept of having an optimum operator:

```
class Opt t where
  optimum :: t -> t -> t
```

The corresponding instances for the types TMin and TMax simply use min or max as optimum:

```
instance Ord v => Opt (TMin v) where
  optimum = min
instance Ord v => Opt (TMax v) where
  optimum = max
```

On the other hand, the product operator will concatenate the partial sequences unless one of them is Nothing, in which case, the result is also Nothing. This is easy to accomplish using the Applicative instance for Maybe. The Semiring instance is then:

```
instance (Semiring sc, Opt sc, Eq sc) => Semiring (BestSolution d sc) where
    sol1@(BestSolution _ sc1) <+> sol2@(BestSolution _ sc2)
       | optimum sc1 sc2 == sc1 = sol1
       | otherwise = sol2
    BestSolution ds1 sc1 <.> BestSolution ds2 sc2 =
       BestSolution ((++) <$> ds1 <*> ds2) (sc1 <.> sc2)
    zero = BestSolution Nothing zero
    one = BestSolution (Just []) one
```

The instance of DPTypes defines combine so that it pairs the score with a list containing only the decision:

```
instance DPTypes sc d sol => DPTypes sc d (BestSolution d sol) where
    combine sc d = BestSolution (Just [d]) (combine sc d)
```

However, we also include another instance for the case when the decisions themselves have type Maybe and we want to collect only those values within a Just, like in the knapsack case.

```
instance DPTypes sc (Maybe d) sol => DPTypes sc (Maybe d) (BestSolution d sol) where
    combine sc d = BestSolution (Just $ maybeToList d) (combine sc d)
```

It is interesting to note the nested use of `combine` so that we can ask for the best solution when maximizing or minimizing by just using the adequate type for the result of `dpSolve`. In the case of the knapsack the line

```
print (dpSolve problem :: TMax Value)
```

prints the maximum value of the knapsack whereas the function

```
decisions :: BestSolution d sc -> [d]
decisions (BestSolution s _) = fromJust s
```

recovers the sequence of decisions from an instance of `BestSolution`, like in the example we will see in Section 9.3.

8 | A REPERTOIRE OF SEMIRINGS

There are other interesting results that can be obtained by changing the semiring. For instance, we may be interested in the number of different solutions to the problem (e.g. how many different knapsacks can be composed). This can be accomplished by defining the `Count` semiring.

```
newtype Count = Count Integer deriving Show

instance Semiring Count where
  Count n <+> Count n' = Count $ n + n'
  Count n <.> Count n' = Count $ n * n'
  zero = Count 0
  one = Count 1

instance DPTypes sc d Count where
  combine _ _ = Count 1
```

Or we may be interested in finding all the possible solutions together with their scores. For this, we define the `AllSolutions` semiring.

```
newtype AllSolutions d sc = AllSolutions [([d], sc)] deriving Show

instance Semiring sc => Semiring (AllSolutions d sc) where
  AllSolutions sols1 <+> AllSolutions sols2 = AllSolutions (sols1 ++ sols2)
  AllSolutions sols1 <.> AllSolutions sols2 =
    AllSolutions [ (ds1 ++ ds2, sc1 <.> sc2)
                 | (ds1, sc1) <- sols1, (ds2, sc2) <- sols2]
  zero = AllSolutions []
  one = AllSolutions [([], one)]

instance DPTypes sc d sol => DPTypes sc d (AllSolutions d sol) where
  combine sc d = AllSolutions [([d], combine sc d)]

instance DPTypes sc (Maybe d) sol => DPTypes sc (Maybe d) (AllSolutions d sol) where
  combine sc d = AllSolutions [(maybeToList d, combine sc d)]
```

An interesting and easy exercise is to write the definition of the semiring `BestSolutions` to hold all the possible solutions that reach the best score. It can be represented by a pair with a score and a list of solutions and the instances are a mixture of those of `BestSolution` and `AllSolutions`.

As a final possibility, we mention that tuples can be used when more than one result is desired, for instance the best solution and the number of them. For this, we introduce the appropriate instances of `Semiring` and `DPTypes`:

```
instance (Semiring s1, Semiring s2) => Semiring (s1, s2) where
  (s1, s2) <+> (s1', s2') = (s1 <+> s1', s2 <+> s2')
  (s1, s2) <.> (s1', s2') = (s1 <.> s1', s2 <.> s2')
  zero = (zero, zero)
  one = (one, one)

instance (DPTypes sc d sol, DPTypes sc d sol') => DPTypes sc d (sol, sol') where
  combine sc d = (combine sc d, combine sc d)
```

It is instructive to realize that the fact that classes can dispatch on the return type is what differentiates the first and second components of these seemingly redundant pairs.

9 | SOLVING OTHER PROBLEMS

Once we have all the pieces in place, it is easy to solve other DP problems.

9.1 | Fibonacci

Although it is a bit of a stretch to consider it as a Dynamic Programming problem, we can easily implement `fibonacci`:

```
fibonacci :: Int -> Integer
fibonacci = dpSolve . fib
  where
    fib :: Int -> DPProblem Int Integer ()
    fib n = DPProblem n (<= 2) (\n -> [(1, (), n-1), (1, (), n-2)])
```

Note the use of the unit type since there are no actual decisions made.

This needs the `Semiring` instance for `Integer`, which is trivial to write:

```
instance Semiring Integer where
  (<+>) = (+)
  (<.>) = (*)
  zero = 0
  one = 1
```

9.2 | Text Segmentation

A naïve approach to text segmentation (i.e. dividing a string like `"helloworld"` into `"hello"` and `"world"`) is to use a mapping from strings to probabilities. Then the task is to find the sequence of words whose concatenation is a given string and such that the product of the probabilities of those words is maximized.

An empty string is a trivial problem and it has probability one. For a nonempty string we must consider each possible prefix that is a key of the dictionary. Then the decision is that prefix; the associated score, its probability; and the subproblem, the rest of the string. We use `inits` and `tails` to find the prefixes and suffixes and a map for the dictionary, and we reuse the `Probability` type as defined in Section 2.3.

The code is then:

```
type Dictionary = Map String Probability

tsDPProblem :: Dictionary -> String -> DPProblem String Probability String
```

```
tsDPProblem d s = DPProblem s isTrivial subproblems
  where
    isTrivial = null
    subproblems s = [ (p, w, s')
                    | (w, s') <- tail $ zip (inits s) (tails s)
                    , Just p <- [M.lookup w d]
                    ]
```

We need to choose an adequate semiring as we want to maximize the product of the probabilities (note that `TMax` maximizes the *sum*). Therefore, we define a new semiring and its corresponding instances:

```
newtype MaxProd v = MaxProd v deriving (Eq, Ord, Show)

instance (Num v, Ord v, Bounded v) => Semiring (MaxProd v) where
  t1 <+> t2 = max t1 t2
  t1@(MaxProd v1) <.> t2@(MaxProd v2)
    | t1 == zero = zero
    | t2 == zero = zero
    | otherwise = MaxProd (v1 * v2)
  zero = MaxProd minBound
  one = MaxProd 1

instance DPTypes sc d (MaxProd sc) where
  combine = const . MaxProd

instance Ord v => Opt (MaxProd v) where
  optimum = max
```

And to use `MaxProd` with probabilities, an instance of `Bounded` must be provided:

```
instance Bounded Probability where
  maxBound = 1
  minBound = 0
```

With an appropriate dictionary, we can easily find the best segmentation:

```
textSegmentation :: Dictionary -> String -> BestSolution String (MaxProd Probability)
textSegmentation d = dpSolve . tsDPProblem d
```

Note that this is one of those cases mentioned in Sections 5 and 7 where there can be problems that have no solution.

9.3 | The Longest Common Subsequence

In the problem of the longest common subsequence (LCS), we are given two lists and want to find the longest list that is a subsequence of both lists, where a list `s` is a subsequence of another list `t` if all the elements of `s` appear in `t` in the same order. For instance, `"ade"` is a subsequence of `"abcdef"`. To frame it as a Dynamic Programming problem, we note that the LCS of two lists when one of them is empty is the empty list. Otherwise if the two lists have the same first element, that element is part of the LCS. Finally, if the lists have different first elements, one of the two has to be dropped, thus giving rise to two subproblems. We can use again `Just x` to signal that `x` is picked while `Nothing` indicates that the element is dropped. In the first case, the only subproblem comprises the tails of both strings; in the second, we have two new subproblems: the tail of one string with the whole of the other string and vice versa.

The function for generating the `DPProblem` is:

```
lcsDPProblem :: Eq a => [a] -> [a] -> DPProblem ([a], [a]) Int (Maybe a)
```

```
lcsDPProblem xs ys = DPProblem (xs, ys) isTrivial subproblems
  where isTrivial (xs, ys) = null xs || null ys
        subproblems (l@(x:xs), r@(y:ys))
          | x == y = [(1, Just x, (xs, ys))]
          | otherwise = [ (0, Nothing, (xs, r))
                        , (0, Nothing, (l, ys))
                        ]
```

The corresponding function to compute the LCS of two strings is:

```
lcs :: String -> String -> String
lcs xs ys = let
       sol = dpSolve $ lcsDPProblem xs ys :: BestSolution Char (TMax Int)
    in decisions sol
```

where it is necessary to explicitly annotate the types since `decisions` (page 12) works the same for maximizations and minimizations.

10 | RELATED WORK

Dynamic Programming is a well known technique. The "text book" approach like in Cormen et al.[1] is to define a suitable table and store the intermediate values there. An example of this approach used in functional programming can be found in Rabhi and Lapalme[2], where they propose to use a ADT for the table. Then, their function `dynamic` receives two parameters, a function that computes a given entry in the table, possibly using other entries, and a pair of indices delimiting the boundaries of the table. The result of `dynamic` is the filled table which is then consulted to retrieve the result. Separate functions are needed to recover the best solution from the table.

A different approach, based on Category Theory is followed in other papers like Kabanov and Vene[7], Hinze and Wu[8], or Hinze[9], where the idea is to define for each problem an adequate data structure to store the intermediate results, which implies an *ad hoc* coalgebra for each problem. This makes it necessary to define a new recursion scheme, the *dynamorphism* to allow the incorporation of such coalgebras. These works are more theoretical and do not consider the recovery of the best solution. This is also the case of Moor[10], where even the implementation of a working program is not considered. More applied but still with the need of an *ex novo* derivation of the storage data structure is Chapter 9 of Bird and de More[11].

By contrast, our proposal is to directly include the memoization within the hylomorphism. A way to interpret this proposal is to consider that memoization is more of an implementation problem: once it is established that the problem is solvable by a hylomorphism, memoization reduces the cost so that the implementation is practical. This way, the user of `dpSolve` only has to describe the way the problem is decomposed and what kind of result is expected by selecting a semiring.

11 | CONCLUSIONS

It is possible to define a very generic function to solve Dynamic Programming problems. This function only needs a description of the problem in terms of how its instances are decomposed into smaller instances and how the solutions to the smaller instances compose to produce the solution of the original instance. Based on that description the function can produce different answers, say the best solution, the score of that solution, or the total number of solutions. Two factors made this possible: the use of the concept of semiring to abstract the operations performed with the solutions and the use of return type polymorphism in Haskell.

ACKNOWLEDGMENTS

Work partially supported by the *Ministerio de Ciencia/AEI/FEDER/EU* through the *MIRANDA-DocTIUM* project (RTI2018-095 645-B-C22).

How to cite this article: David Llorens, Juan Miguel Vilar, Easily Solving Dynamic Programming Problems in Haskell by Memoization of Hylomorphisms, *Software: Practice and Experience*. 2020; *50*i: 2193–2211. DOI: https://doi.org/10.1002/spe.2887

APPENDIX

A `HASTRIE` INSTANCES

The derivation of `HasTrie` instances for `Generic` types (as defined in `GHC.Generics`), which covers the algebraic datatypes is quite mechanical. The pragmas needed for this are `DeriveGeneric`, `StandaloneDeriving`, `TypeFamilies`, and `TypeOperators`. With them, an instance declaration simply defines a new type and three functions. The idea is that the type acts as a bridge between the generic version of the values and their concrete instantiation.

For example, the instances for `Item` and `KSProblem` are:

```
deriving instance Generic Item

instance HasTrie Item where
  newtype (Item :->: b) = ItemTrie { unItemTrie :: Reg Item :->: b }
  trie = trieGeneric ItemTrie
  untrie = untrieGeneric unItemTrie
  enumerate = enumerateGeneric unItemTrie

deriving instance Generic KSProblem

instance HasTrie KSProblem where
  newtype (KSProblem :->: b) = KSTrie { unKSTrie :: Reg KSProblem :->: b }
  trie = trieGeneric KSTrie
  untrie = untrieGeneric unKSTrie
  enumerate = enumerateGeneric unKSTrie
```

References

1. Cormen TH, Leiserson CE, Rivest RL, Stein C. *Introduction to Algorithms*. MIT Press. 3 ed. 2009.

2. Rabhi F, Lapalme G. *Algorithms: A Functional Programming Approach*. International computer science seriesAddison-Wesley . 1999.

3. GHC The Glasgow Haskell Compiler. https://www.haskell.org/ghc; 2020. Accessed: 2020-04-18.

4. Meijer E, Fokkinga MM, Paterson R. Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. In: Hughes J., ed. *Functional Programming Languages and Computer Architecture (FPCA 1991)*. 523 of *LNCS*. ; 1991: 124–144.

5. Hinze R. Memo functions, polytypically!. In: Workshop on Generic Programming 2000. ; 2000; Ponte de Lima, Portugal: 17–32.

6. Elliott C. Data.MemoTrie. https://hackage.haskell.org/package/MemoTrie; 2016. Accessed: 2019-11-04.

7. Kabanov J, Vene V. Recursion Schemes for Dynamic Programming. *Lecture Notes in Computer Science* 2006; 4014 LNCS: 235–252.

8. Hinze R, Wu N. Histo- and dynamorphisms revisited. *Proceedings of the 9th ACM SIGPLAN workshop on Generic programming - WGP '13* 2013: 1.

9. Hinze R, Wu N, Gibbons J. Conjugate Hylomorphisms – Or. In: ACM Press; 2015; New York, New York, USA: 527–538.

10. Moor dO. *Categories, Relations and Dynamic Programming*. No. 98 in Categories, Relations and Dynamic ProgrammingOxford University Computing Laboratory, Programming Research Group . 1992.

11. Bird R, de Moor O. *Algebra of Programming*. USA: Prentice-Hall, Inc. . 1997.