A Tabulation Transformation Tactic Using Haskell Arrays

Cristóbal Pareja-Flores*, Ricardo Peña* and J. Ángel Velázquez-Iturbide†

Abstract

In this paper we propose a transformation tactic that provides general tabulation of functional algorithms. This tabulation tactic can be applied to dependency graphs in which variable size cuts can be defined. The tactic can be considered a generalization of the *tupling* tactic proposed by other authors. Tables are dynamically created and their sizes determined at execution time. The tactic is greatly simplified by the use of the data type *array* present in the functional language Haskell. According to the size of the tables used, two forms of tabulation are distinguished, respectively named *total* and *partial tabulation.* Some significative examples are developed using the tactic, including *dynamic programming* problems. Finally, we apply the loop inversion tactic to the partially tabulated algorithms to show that, in many cases, these algorithms can be transformed into tail recursive versions, similar to their imperative counterparts.

Keywords: functional programming, tabulation, memoization, tupling, program transformation, dynamic programming, Haskell.

1 Introduction

Tabulation is a well-known technique to avoid redundant computations of recursive functions, that is, to avoid that a recursive function is invoked more than once to solve the same subproblem. This situation is typical in problems solved by the *dynamic programming* method. The tabulation technique is based on storing in a table the result of an invocation so that the value of each subproblem is computed only once.

The subject has been studied by several authors. In[Bir80] the main elements of the technique are identified: first, the *dependency* (acyclic directed) *graph* of recursive calls developed during a function invocation is studied; then, an order to solve the subproblems is established in such a way that, when a subproblem is needed, it is already solved and stored; finally, an iterative program solves the subproblems following such an order. In some complex cases, the author recommends to use the original recursive function, together with a global table providing space enough to keep the results of *all* the subproblems. This is called *exact tabulation* in his paper, and we will refer to it here as *total tabulation.*

In[Pet84] a transformation technique to avoid redundant computations for a class of functions is proposed. The technique is called *tupling* and consists of defining a sequence of *cuts* in the dependency graph of the original function in such a way that every cut (a cut is a tuple of invocations) can be computed in terms of the precedent one. Then an auxiliary function, later

[1]Departamento de Informática y Automática, Universidad Complutense de Madrid, 28040 Madrid, Spain. e-mails: {cpareja,ricardo}@dia.ucm.es

[2]Departamento de Lenguajes y Sistemas Informáticos, Universidad Politécnica de Madrid, 28660 Boadilla del Monte, Madrid, Spain. e-mail: avelazquez@fi.upm.es

transformed into a linear recursive function, is defined as a tuple of invocations of the original function. This one can be easily expressed in terms of the tupled one. For the strategy to be applied, all the cuts are required to have the same number of nodes. The tuple representing the cut plays the role of a table tabulating the results of the previously solved subproblems. In the terminology we use in this paper, tupling is equivalent to *partial tabulation* using a fixed size table.

Other relevant works on redundant computation removal, tabulation, and tupling are[Hug85, Chi90, Kho90]. *Memoization* (e.g. see[Hug85, FH88, Kho90]) corresponds to total tabulation implemented by a compiler.

In this paper we propose a transformation tactic to produce a tabulated version of a recursive function with redundant calls, in which sizes of cuts can be determined at execution time. Our motivation for this work is mainly methodological: we wish to study more general transformations than tupling. In particular, the classical imperative dynamic programming algorithms should be systematically obtained by applying our techniques.

One extreme case of our technique is *total tabulation*, used when the dependency graph does not admit sequences of cuts. The other extreme is *tupling*, used when cuts with a fixed (and small) number of nodes can be defined in the dependency graph. In the general case, we have *partial tabulation* in which a variable size table is used to record the results of a cut. Other differences with respect to some of the previously mentioned approaches are: 1) we do not leave in any respect the functional paradigm, and 2) the proposed transformations are carried out by the programmer (not by the compiler).

Both purposes are eased by the use the data structure *array* present in the functional language Haskell[HJW92]. Arrays in Haskell may be created, in a monolithic way, by using the function `mkArray` that has as one of its parameters a *creation function* mapping indexes into values. In our tactic, the original redundant function is transformed into the creation function of an array, and array entries are made dependent on other entries of a different (or of the same) array, in a recursive way. Updating in place will be almost never required in this tabulation technique, so the compiler will only maintain a single copy of the current array and, consequently, the constant access time of imperative arrays is retained by the approach.

The organization of the paper is as follows: after this introduction, in section 2 we define and prove correct a schematic rule to transform the recursive version of a function into a totally tabulated version. In section 3, the concept of *cut* and the partial tabulation tactic are introduced. This tactic allows to reduce the amount of table space required, since the size of the table depends in general of the current cut being processed. Both tabulation tactics are based on the rules of the *fold/unfold* system[BD77]. In section 4, we show the application of the inversion tactic to partially tabulated algorithms to produce tail recursive versions. In section 5, two significative examples are developed using the tactics. Finally, section 6 contains some conclusions.

2 Total tabulation

In Haskell[HJW92], the data type `Array` is defined as follows:

```
    data (Ix a) => Array a b = MkArray (a,a) (a -> b)
```

The class `Ix` is that of array indexes. It is a subclass of the class `Ord` of types with an order relation, and a number of additional operations are required for it:

```
range   :: (a,a) -> [a]        --lists all indexes between bounds
index   :: (a,a) -> a -> Int  --converts an index into a position
inRange :: (a,a) -> a -> Bool --check an index to be in range
```

However, arrays are considered as abstract data types and the constructor `MkArray` is not exported. The programmer may create the array by using the predefined function `array :: (Ix a) => (a,a) -> [Assoc a b] -> Array a b` which needs an *association list*. The data type `Assoc` defines tuples of the form `i := v`, where `i` is an index and `v` is the value we wish to associate, in the new array, to index `i`. Alternatively, an array may be created using the function

```
mkArray  :: (Ix a) => (a,a) -> (a -> b) -> Array a b
```

as defined in[HF92], which takes as arguments the array bounds and a function from indexes to values that we will call the *creation function*. In any of these ways, arrays are dynamically created. Once created, they can be consulted or updated by using the predefined operations:

```
(!)   :: (Ix a) => Array a b -> a -> b
(//)  :: (Ix a) => Array a b -> [Assoc a b] -> Array a b
```

A reasonable implementation of the array feature will, of course, map an array into consecutive memory locations so that operator `(!)` could execute in constant time. Also, compile time analysis techniques are expected to detect those situations in which arrays can be safely updated in place without generating fresh copies of them.

If `a1, a2` are instances of the class `Ix`, so is the tuple `(a1,a2)` formed by them. The `range` function for tuples is defined in such a way that array elements are enumerated by rows. For example, `range ((0,4),(1,5))` produces the list:

```
[(0,4),(0,5),(1,4),(1,5)]
```

We make note that requiring the parameters of the function to belong to class `Ix` is not as a big restriction as it may seem. Should the parameters be more complex, hashing techniques could in most cases be used to map them into a subrange of integers. In this cases, parameters would be kept in the array implementing the hash table, and the only requirement for them would be to belong to class `Eq`.

The following definition introduces a schematic rule for total tabulation of redundant functions, using arrays.

Definition 1 If `f p1...pn = rhs` defines a recursive function `f`, `t1` is the type of `p1`,..., `tn` is the type of `pn`, and `(Ix t1,...,tn)` holds, we call the *totally tabulated* version of `f` to the following set of definitions:

```
f p1...pn = fTable ! (p1,...,pn)
            where fTable         = mkArray b f'
                  f' (j1,...,jn) = rhs'
```

where `b` has the form `((l1,...,ln),(u1,...,un))` and, for all `i`, `li`, `ui` are respectively the lower and upper bounds of argument `pi`. They depend, in general, on the parameters `p1...pn`, hence the definition of `fTable` in a local clause where the arguments `p1...pn` are visible. The expression `rhs'` is obtained from `rhs` by substituting `ji` for `pi`, for all `i`, and the subexpression `fTable!(e1,...,en)` for any recursive call in `rhs` of the form `f e1...en`. We denote this fact by `rhs'`$\overset{\text{def}}{=}$ `rhs[j1,...,jn,fTable!(e1,...,en)/p1,...,pn,f e1...en]`.

□

Proposition 2 Given a recursive definition for a function `f` and the definition of its totally tabulated version, both compute the same function provided that $\forall$`i.li` $\leq$ `pi` $\leq$ `ui` is an invariant of `f`. □

The term *invariant* is used here in the following sense: it is a property that can be shown holds for any internal call to `f` assuming it holds for `f`.

We can prove this proposition, i.e. that the transformation process preserves the semantics of the original definition, by equational reasoning. In this proof we are using the fold/unfold system of[BD77], whose rules can be summarized as follows:

Definition rule A new equation `lhs = rhs` is introduced so that `lhs` is not an instance of a left hand side of any other equation.

Instantiaton rule An equation `lhs = rhs` is derived from a previous one, where `lhs` and `rhs` are respectively instances of its left and right hand sides such that `lhs` is a pattern expression.

Unfolding rule In the right hand side of an equation, a subexpression `e` which is an instance of the left hand side of another equation is replaced by the corresponding instantiated right hand side `e'`.

Folding rule In the right hand side of an equation, a subexpression `e` which is an instance of the right hand side of another equation is replaced by the corresponding instantiated left hand side `e'`.

Abstraction rule All the ocurrences of subexpressions `e1,...,en` in a subexpression `e` of the right hand side of an equation are abstracted in a `where` clause, resulting the new expression `e[z1,...,zn/e1,...,en] where (z1,...,zn) = (e1,...,en)`, where `z1,...,zn` are fresh variables.

Law rule It is an equality rewriting rule like the unfolding and folding rules, but the equation referenced is an arbitrary algebraic law `e = e'`.

We need to use the following obvious algebraic law for arrays:

$$\texttt{mkArray b f!i = f i} \quad \text{provided} \quad \texttt{inRange b i} \quad \text{holds} \tag{1}$$

Then, the following derivation proves the proposition:

$$\begin{array}{ll} & \texttt{f p1...pn} \\ \equiv & \texttt{fTable!(p1,...,pn)} \quad \{\text{definition rule}\} \\ \equiv & \texttt{mkArray b f'!(p1,...,pn)} \quad \{\text{unfolding rule}\} \\ \equiv & \texttt{f' (p1,...,pn)} \quad \{\text{law rule, using law 1}\} \\ \equiv & \texttt{rhs'}[\texttt{p1},...,\texttt{pn}/\texttt{j1},...,\texttt{jn}] \quad \{\text{unfolding rule}\} \\ \equiv & \texttt{rhs} \quad \{\text{folding rule on } \texttt{f} \text{ assuming the invariant holds}\} \end{array}$$

Example 3 The following recursive function defines the combinatory number $\binom{m}{n}$:

```
comb m 0           = 1
comb m n | m == n = 1
         | m > n  = comb (m-1) (n-1) + comb (m-1) n
```

Applying the schematic rule given by proposition 2, we transform `comb` into the following totally tabulated version:

```
comb m n = combTable!(m,n) where
             combTable = mkArray ((0,0),(m,n)) comb'
             comb' (i,0)          = 1
             comb' (i,j) | i == j = 1
                         | i > j  = combTable!(i-1,j-1) +
                                    combTable!(i-1,j)
```

□

3 Partial tabulation

In most cases, totally tabulated algorithms are too expensive in memory space. Our partial tabulation tactic produces a tabulated algorithm using less space. In the tupling tactic, first the redundancy structure of a given function is analyzed, and then an auxiliary function —defined as a tuple of invocations of the original one— is transformed into a linear recursive function. In our approach, the strategy is similar. The main difference is that we allow cuts to be of variable size.

The *dependency graph*[Bir80] for a concrete function application is an acyclic graph obtained from the tree of recursive calls by identifying nodes with identical arguments. A *cut* is a set of nodes of the dependency graph such that, were these nodes —and their incident edges— deleted from the graph, the result would be two disjoint graphs. A sequence of cuts $c_n, c_{n-1}, \ldots, c_0$ is *progressive* if, for all i, $1 \le i \le n$, every node in cut c_i can be computed from nodes in c_i or in c_{i-1}. To be more precise, we reproduce the definition given by Pettorossi[Pet84]:

Definition 4 A sequence of cuts $c_n, c_{n-1}, \ldots, c_0$ is *progressive* if,

$$\begin{aligned}&\forall i, 1 \le i \le n.(c_i \ne c_i \cap c_{i-1} \ne c_{i-1})\\ &\wedge(\forall m \in c_i - c_{i-1}.\exists n \in c_{i-1}.m{\rightarrow}n) \wedge (\forall n \in c_{i-1} - c_i.\exists m \in c_i.m{\rightarrow}n)\end{aligned}$$

where $m{\rightarrow}n$ stands for "there exists a directed edge from node m to node n". □

3.1 The partial tabulation tactic

For the sake of clarity, let us assume that the recursive function is defined by the single equation:

$$\texttt{f p1...pn} = \texttt{rhs} \tag{2}$$

where `rhs` contains references to `f`. We will call `res` to the type of `rhs`.

The first step of this tactic is, like in the tupling technique, to define a progressive sequence of cuts in the dependency graph. Now, however, we remove the limitation that all the cuts must have a fixed number of nodes.

The second step consists of the proper transformation, similar to that developed within the tupling tactic. First —by applying the definition rule (see section 2)— we introduce an *eureka* equation with the following format:

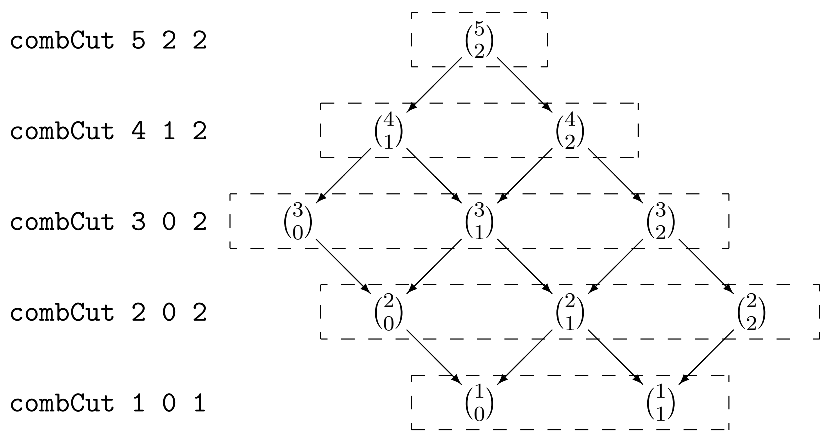

Figure 1: Cuts for the call `combCut 5 2 2`

```
fCut q1...qr = mkArray b f'  where f' i = f e1...en
```

where `fCut q1...qr` is the table to store the results of a cut, `b::(a,a)` are its bounds which, in general, depend on `q1,...,qr`, `f' :: a -> res` is its creation function, and `e1...en` are expressions depending on `q1,...,qr` and `i`. Since the cut table will store values computed by `f`, one can expect that `f'` will be initially defined in terms of `f` (in some cases `f'` even coincides with `f`).

For any cut and using law 1, we have the property:

$$\texttt{fCut q1...qr ! i} \equiv \texttt{f e1...en} \;\; \text{provided} \;\; \texttt{inRange b i} \qquad (3)$$

In particular, for the top level cut and for some actual parameters `s1,...,sr` and index `s` such that `inRange btop s`, where `btop` depends on `s1,...,sr`, we have:

$$\texttt{fCut s1...sr ! s} \equiv \texttt{f p1...pn} \qquad (4)$$

The next steps consist of transforming the defined function into a linear recursive function by replacing all references to `f` in the definition of `f'` by recursive calls to `fCut`. For this purpose, we usually instantiate and unfold the definition of `f'`, we use property 3 to fold references to `f` into references to a cut table, and we abstract references to the new cut table.

Finally, if we have succeeded in the previous steps, we eliminate equation 2 and replace it by a right to left version of property 4:

```
f p1 ... pn = fCut s1 ... sr ! s
```

Example 5 In example 3 the result for `comb m n` depends, for all `m`> 0, on two values belonging to the previous row of the Pascal triangle. So, the rows in the Pascal triangle form a progressive sequence of cuts. In fact, analyzing the dependency graph of the function, it can be seen that only a portion of each row in the triangle is needed. We define a cut as the values from a lower bound `j1` to an upper bound `j2` of row `i`. Figure 1 illustrates this idea. So, we give the following *eureka* equation:

$$\mathtt{combCut\ i\ j1\ j2} = \mathtt{mkArray\ (j1,j2)\ comb'\ \ where\ \ comb'\ k} = \mathtt{comb\ i\ k} \tag{5}$$

and obtain the following two laws:

$$\begin{aligned} \mathtt{combCut\ i\ j1\ j2\ !\ \ k} &\equiv \mathtt{comb\ i\ k} \quad \forall \mathtt{k} \in \{\mathtt{j1} \ldots \mathtt{j2}\} \\ \mathtt{combCut\ m\ n\ n\ !\ \ n} &\equiv \mathtt{comb\ m\ n} \end{aligned}$$

The transformation based on equation 5 produces the following tabulated version of `comb`:

```
comb m n = combCut m n n ! n
combCut   0    0  0 = mkArray (0,0) (const 1)
combCut (i+1) j1 j2 = mkArray (j1,j2) comb'
   where comb' k | k==0 || k==(i+1) = 1
                 | otherwise        = upRow ! (k-1) + upRow ! k
         upRow = combCut i (max 0 (j1-1)) (min j2 i)
```

The transformation is straightforward, with the exception of the limits of the cuts, which require additional reasoning.

□

4 Loop inversion

In this section, we apply the tactic known as *loop inversion*[Boi92, Par90] to our partial tabulated algorithms. This tactic transforms a non-tail linear recursive function into a tail recursive one corresponding to the ascending loop of the first function. For this tactic to be applied, some conditions must hold. The transformation is contained in the following

Proposition 6 Let a non tail linear recursive function `f` be defined as:

```
f x | p x = q x
    | otherwise = x ⊕ f (t x)
```

where $\oplus$ is a (not necessarily associative) binary operator. Then, if the following conditions hold,

1. It is possible to determine a unique base case, $\mathtt{x_{triv}}$ for every initial value $\mathtt{x_{init}}$.
2. The descent function `t` admits inverse function. We denote it by `t'`.

we have:

```
f x_init = g x_triv (q x_triv) where
        g x fx | x == x_init= fx
               | otherwise = let x' = t' x in g x' (x' ⊕ fx)
```

□

Note that $\mathtt{x_{triv}}$ and `t'` may depend on the argument $\mathtt{x_{init}}$. If they do, the dependencies should be through constant-time functions for this proposition to be useful.

The tail recursive algorithms we obtain by loop inversion are similar to the iterative algorithms obtained by tabulation in the imperative paradigm.

Example 7 The application of loop inversion to our partial tabulated version of combinatory numbers yields the following program:

```
comb m n = combCut m n n ! n
combCut 0 0 0  = mkArray (0,0) (const 1)
combCut m n n  = combLoop 1 0 1 (mkArray (0,1) (const 1)) where
   combLoop i j1 j2 irow
      |  i == m   = irow
      | otherwise = combLoop i' j1' j2' (mkArray (j1',j2') f')
                    where i'  = i+1
                          j1' = max (n-m+i') 0
                          j2' = min n i'
                          f' k | k == 0 || k == i' = 1
                               | otherwise         = irow!(k-1) +
                                                     irow!k
```

□

5 Cases study

5.1 0/1-knapsack problem

This is the well-known problem[AHU83, HS90] of finding the maximum profit we can get by introducing some subset of n objects with given weights and profits in a knapsack with fixed capacity c. For simplicity, we assume the existence of two global arrays `ws, ps :: Array Int Int`, of length `n`, respectively containing the weights and profits of the n objects. The initial recursive algorithm follows:

```
knapsack   0   c = 0
knapsack (n+1) c
   | c<ps!(n+1) = knapsack n c
   | otherwise  = max (knapsack n c)
                      (knapsack n (c-ws!(n+1)) + ps!(n+1))
```

where the solution to our problem is given by the call `knapsack n c`.

In the analysis phase, we observe that its dependency graph is a submatrix of an $n \times c$ matrix, whose items represent the values of `knapsack i j`, for $0 \leq i \leq n$ and $0 \leq j \leq c$. Therefore, we foresee an array with these bounds for the total tabulation version. By applying proposition 2, we obtain:

```
knapsack n c = knapsackTable!(n,c) where
  knapsackTable = mkArray ((0,0),(n,c)) knapsack'
  knapsack'( 0 ,j) = 0
  knapsack'(i+1,j) | j<ws!(i+1) = knapsackTable!(i,j)
                   | otherwise  = max (knapsackTable!(i,j))
                                      (knapsackTable!(i,j-ws!(i+1))
                                       + ps!(i+1))
```

We can also partially tabulate the algorithm by choosing cuts corresponding to rows of the total table. The cut for row `i` fixes an object index `i` and let the capacity to vary between 0 and `c`. The definition representing this idea is:

```
knapsackCut i c = mkArray (0,c) f'  where  f' j = knapsack i j
```

Instantiating laws 3 and 4, we have the laws:

$$\texttt{knapsackCut i c!j} \equiv \texttt{knapsack i j} \quad \forall j \in \{0 \dots c\} \tag{6}$$

$$\texttt{knapsackCut n c!c} \equiv \texttt{knapsack n c} \tag{7}$$

To transform `f'`, we instantiate equation 6 according to the definition of `knapsack`:

```
knapsackCut 0 c = mkArray (0,c) f' where
  f' j = knapsack 0 j ≡ 0    {by unfolding}
knapsackCut (i+1) c = mkArray (0,c) f' where
  f' j = knapsack (i+1) j
```

Now, the transformation of the `f'` local to the scope of `knapsackCut (i+1) c` results in:

```
f' j | j < ws!(i+1)   = knapsack i j
                      ≡knapsackCut i c!j   {by law 6}
     | otherwise      = max (knapsack i j)
                            (knapsack i (j-ws!(i+1)) + ps!(i+1))
                      ≡max (knapsackCut i c!j)   {by law 6}
                            (knapsackCut i c!(j-ws!(i+1)) + ps!(i+1))
```

Finally, we apply the abstraction rule and obtain:

```
knapsack n c = knapsackCut n c ! c
knapsackCut   0   c = mkArray (0,c) (const 0)
knapsackCut (i+1) c = mkArray (0,c) f' where
   f' j | j < ws!(i+1) = previousRow!j
        | otherwise    = max (previousRow!j)
                             (previousRow!(j-ws!(i+1)) + ps!(i+1))
                           where previousRow = knapsackCut i c
```

Inverting the function `knapsackCut` is straightforward, since the descent function is `\i -> i-1`.

```
knapsack n c = knapsackCut n c ! c
knapsackCut n c = knapsackLoop 0 (mkArray (0,c) (const 0)) where
  knapsackLoop i row
    | i == n    = row
    | otherwise = knapsackLoop i' (mkArray (0,c) f') where
                   i' = i+1
                   f' j | j < ws!i' = row!j
                        | otherwise = max (row!j)
                                          (row!(j-ws!i') + ps!i')
```

5.2 Fibonacci sequence

We include this well-known example to show that tupling can be considered as a particular case of partial tabulation.

```
fib   0   = 1
fib   1   = 1
fib (n+2) = fib n + fib (n+1)
```

Here, we are only interested in the partially tabulated version. The obvious cut is a pair of consecutive Fibonacci numbers.

```
fibCut n = mkArray (n,n+1) fib'  where  fib' i = fib i
```

so the following law holds:

$$\texttt{fibCut n ! i} \equiv \texttt{fib i} \tag{8}$$

for $i = n$ and $i = n + 1$. Now, we instantiate `fibCut n` for two patterns of `n` and `fib'` for the two values of `i`, namely `n+1` and `n+2`, in the second equation of `fibCut`, resulting in:

```
fibCut   0    = mkArray (0,1) fib' where fib' i = 1
fibCut (n+1)  = mkArray (n+1,n+2) fib'
    where fib' i | i==n+1 =   fib (n+1)
                          ≡   fibCut n !  (n+1)           {by law 8}
                 | i==n+2 =   fib (n+2)
                          ≡   fib n + fib(n+1)            {by unfolding}
                          ≡   fibCut n!n + fibCut n!(n+1) {by law 8}
```

Finally, by the abstraction rule, we obtain:

```
fib n = fibCut n ! n
fibCut   0   = mkArray (0,1) (const 1)
fibCut (n+1) = mkArray (n+1,n+2) fib' where
               fib' i | i==n+1 = prevCut ! (n+1)
                      | i==n+2 = prevCut ! n + prevCut ! (n+1)
                        where prevCut = fibCut n
```

In this way, `fibCut (n+1)` is defined in terms of `fibCut n`. This program can be straightforwardly transformed into the well known tupled version:

```
fib n = fst (fibPair n)
fibPair   0   = (1,1)
fibPair (n+1) = (v,u+v) where (u,v) = fibPair n
```

6 Discussion

We may distinguish three classes of tabulation for functional algorithms. Tupling is characterized by the existence of a progressive sequence of cuts of fixed size. Partial tabulation is similar, but cuts are allowed to have variable size. Finally, total tabulation is a brute-force tabulation solution to be used where progressive sequences of cuts cannot be found. Both partial and total tabulation are relatively simple to apply, although some nontrivial steps may be needed in some problems (cut limits, inverses of descent functions, trivial cases). These tactics transform the redundant algorithm in different ways: total tabulation provides a fixed schematic transformation rule whereas partial tabulation begins with an eureka definition and follows with its subsequent fold/unfold transformation.

The use of Haskell arrays allows a concise presentation of both tabulation tactics and tabulated algorithms. However, by using lists, the main ideas are as well applicable to languages lacking arrays. Of course, some efficiency may be lost. If they provides list comprehensions, the transformation and the final algorithms are very similar.

An interesting aspect regarding partially tabulated algorithms is that they are linear recursive and can usually be inverted. So, the consecutive application of the partial tabulation and of the function inversion tactics produces tail recursive algorithms. These algorithms can be straightforwardly transformed into imperative ones.

We have applied tabulation to a number of redundant algorithms, including several dynamic programming ones. Some of them have been included in the paper; others are the minimal triangulation problem[AHU83] and the multistage graph problem[HS90].

Two last comments about the influence of lazy evaluation in tabulated algorithms follow. Lazy evaluation does not compute the values of `f'` until they are required. As a consequence, the resulting totally tabulated algorithms are equivalent to the corresponding imperative memoized ones[Hug85, FH88, Kho90] except for the fact that the first are functional.

A second remark on lazy evaluation is related to space efficiency. With eager evaluation, the memory space required for inverted partially tabulated algorithms is an order of magnitude less than that for totally tabulated ones. Cuts are computed in sequence, so only two cuts —but not the whole table— are required to be simultaneusly in memory. With lazy evaluation, computations for all the cuts are interleaved and, in tests carried out by the authors, the space improvement is smaller than somebody may expect. Nevertheless, as remarked in the introduction, the actual gain comes from the transformation of the inverted versions to the imperative paradigm.

7 Acknowledgements

This work was partially supported by the spanish agency CICYT, projects TIC91-0111 and TIC95-0967-C02.

References

[AHU83] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1983.

[BD77] R. M. Burstall and J. Darlington. A Transformation System for Developing Recursive Programs. *Journal of the ACM*, 24(1):44–67, January 1977.

[Bir80] R. S. Bird. Tabulation Techniques for Recursive Programs. *Computing Surveys*, 12(4):403–417, December 1980.

[Boi92] E. A. Boiten. Improving Recursive Functions by Inverting the Order of Evaluation. *Science of Computer Programming*, 18:139–179, 1992.

[Chi90] Wei-Ngan Chin. Automatic methods for program transformation. Ph. D. Thesis, Imperial College, University of London, 1990.

[FH88] A. J. Field and P. G. Harrison. *Functional Programming.* Addison-Wesley, 1988.

[HF92] P. Hudak and J. H. Fasel. A Gentle Introduction to Haskell. *ACM SIGPLAN Notices*, 27(5), May 1992.

[HJW92] P. Hudak, S. L. Peyton Jones, and P. Wadler. Report on the Functional Programming Language Haskell. version 1.2. *ACM SIGPLAN Notices*, 27(5), May 1992.

[HS90] E. Horowitz and S. Sahni. *Fundamentals of Data Structures in PASCAL.* Computer Science Press, 3rd edition, 1990.

[Hug85] J. Hughes. Lazy memo-functions. In *Conf. on Functional Programming and Computer Architecture. Lecture Notes in Computer Science 201, Springer Verlag*, 1985.

[Kho90] H. Khoshnevisan. Efficient Memo-Table Management Strategies. *Acta Informatica*, 28:43–81, 1990.

[Par90] H. A. Partsch. *Specification and Transformation of Programs.* Springer-Verlag, 1990. Capítulos 4 y 6.

[Pet84] A. Pettorossi. A Powerful Strategy for Deriving Efficient Programs by Transformation. In *ACM Symposium on Lisp and Functional Programming, Austin,Texas, 6-8 Aug.*, pages 273–281, 1984.