Inverting Dynamic Programming

Nikolay Shilov

A.P. Ershov Institute of Informatics Systems, Russian Academy of Sciences
Lavren'ev av. 6, 630090 Novosibirsk, Russia
shilov@iis.nsk.su
http://persons.iis.nsk.su/en/person/shilov

Abstract. We suggest and discuss a formalization of Dynamic Programming. A methodological novelty consists in an *explicit* treatment (interpretation) of ascending Dynamic Programming as least fix-point computation (according to Knaster-Tarski fix-point theorem). This interpretation leads to a uniform approach to classical optimization problems as well as to problems where optimality is not explicit (Cocke - Younger - Kasami parsing algorithm for example) and to problem of algorithm inversion (i.e. computing inverse function).

Keywords: Dynamic programming, Tarsky-Knaster fixpoint theorem, algorithm inversion

1 Introduction

We would like to continue study of algorithm inversion started in [11]. The cited paper [11] is about a fake coin puzzle to be solved in three programming paradigms: logic, functional and imperative. It can be considered as a case study of algorithm inversion, since we start with logic algorithm, that answers the question "*Is balancing M times sufficient for detecting the fake coin?*", and finishes with imperative algorithm, that effectively computes the minimal number of balancing that is sufficient for detection the fake; functional paradigm is used for developing an intermediate functional algorithm that also computes the minimal number of balancing, but inefficiently. Basically, the functional and the imperative solutions of the puzzle are two (recursive and iterative) versions of dynamic programming algorithm that inverts a logical program. In the present paper we generalize background ideas to inversion of recursive dynamic programming.

1.1 Dropping Bricks from a High Tower

Let us start with the following *Dropping Bricks Puzzle*[1].

[1] When draft of this paper was ready, Prof. Teodor Zarkua (St. Andrew University of Georgian Patriarch) informed the author that the problem is known already and has been used for programming contests (check, for example, the problem at URL http://acm.timus.ru/problem.aspx?space=1&num=1223). Recently a variant of the problem has been added to Wikipedia article Dynamic Programming (available at http://en.wikipedia.org/wiki/Dynamic_programming#Egg_dropping_puzzle).

> Let us characterize mechanical stability (strength) of a brick by an integer h that is equal to the height (in meters) that is safe for the brick to fall down, while height $(h + 1)$ meters is unsafe (i.e. the brick breaks). You have to define stability of bricks of a particular kind by dropping them from different levels of a tower of H meters. (You may assume that mechanical stability does not change after safe fall of a brick.) How many times do you need to drop bricks for it, if you have 2 bricks in the stock? What is the optimal number (of droppings) in this case?

Basically the question that we need to answer is how to compute the optimal number of droppings G_H, if the height of the tower is H, and you have 2 bricks in the stock. In the next subsection we sketch descending Dynamic Programming solution of the above problem as a gentle introduction to Dynamic Programming approach to optimization, design its implementation in terms of functional pseudo-code and conclude with historic remarks.

The rest of the paper is organized as follows. In the next section 2 we introduce (what we call) a *scheme of recursive Dynamic Programming* and discuss in brief how to improve efficiency of recursive Dynamic Programming by memoization. In the section 3 we convert recursive Dynamic Programming into the iterative form and interpret Dynamic Programming as a computation of the least fix-point of an appropriate monotone functional. In turn we get an opportunity to design, specify and verify a *unified template for ascending Dynamic Programming*. Two examples of the template specialization are presented in section 4, one of which is context-free parsing. In the section 5 we suggest an approach how to invert algorithms that are based on Dynamic Programming. We discuss some concluding remarks in the last section 6.

1.2 Recursive Method for Optimization Problems

Dropping Bricks Puzzle is a particular and explicit example of optimization problems. Originally Dynamic Programming has been designed as a recursive search (or construction) of an optimal program (or plan) that remains optimal at every stage. For example let us consider below the puzzle.

Any optimal method (to define mechanical stability) should start with some step (command) that prescribes to drop the first brick from some particular (but optimal) level h. Hence the following equality should hold for this particular h:

$$G_H = 1 + \max\{(h-1), G_{H-h}\},$$

where (in the right-hand side)

1. '1+' corresponds to the first dropping,
2. $(h - 1)$ corresponds to the case when the first brick breaks after the first dropping (and we have to drop the remaining second brick from levels 1, 2, ... $(h - 1)$ in the sequence),
3. G_{H-h} corresponds to the case when the first brick is safe after the first dropping (and we have to define stability by dropping pair of bricks from $(H - h)$ levels in $[(h + 1)H]$),

4. 'max' corresponds to the worst in cases 2 and 3 above.

Since the particular value h is *optimal*, and *optimality* means *minimality*, hence the above equality transforms to the next one:

$$G_H = \min_{1\le h\le H}(1+\max\{(h-1), G_{H-h}\}) = 1 + \min_{1\le h\le H}\max\{(h-1), G_{H-h}\}.$$

Besides we can add one obvious equality $G_0 = 0$.

One can remark that sequence of integers G_0, G_1, ... G_H, ..., that meet these two equalities, is unique, since G_0 is defined explicitly, G_1 is defined by G_0, G_2 — by G_0 and G_1, G_H — by G_0, G_1, ... G_{H-1}. Hence it is possible to move from the sequence G_0, G_1, ... G_H, ..., to a function $G : \mathbb{N} \to \mathbb{N}$ that maps every natural H to G_H and satisfies the following system of functional equations for the *objective function* G:

$$\begin{cases} G(0) = 0 \\ G(H) = 1 + \min_{1\le h\le H}\max\{(h-1), G(H-h)\} \end{cases}.$$

This system has unique solution as it follows from the uniqueness of the sequence G_0, G_1, ... G_H, ... Hence this system can be adopted as a recursive definition of a function, i.e. a recursive algorithm presented by its functional pseudo-code. This is an example of the first (historically) face of Dynamic Programming — a recursive method for optimization problems.

Dynamic Programming was introduced as a recursive method for optimization problems by Richard Bellman in 1950s [5]. At this time the noun *programming* had nothing in common with more recent computer programming, but meant *planning* (compare: *linear programming*). The adjective *dynamic* points out that Dynamic Programming is related to *change of state* (compare: *dynamic logic*, *dynamic system*). Functional equations for the objective function (like in the above) are called after Richard Bellman *Bellman equations* as well as the following *Bellman Principle of Optimality* (that we have use already): *an optimal program (or plan) remains optimal at every stage.*

2 Recursion & Memoization vs. Dynamic Programming

If to analyze the recursive Dynamic Programming methodology accumulated in Bellman Principle and the above recursive solution for Dropping Bricks Puzzle, then it is possible to suggest the following scheme of recursive Dynamic Programming.

Definition 1. *Let scheme of recursive Dynamic Programming be the following recursive program scheme [7,8]*

$$G(x) \;=\; if\ p(x)\ then\ f(x)\ else\ g(x,\ (G(t_i(x)),\ i \in [1..n(x)])), \qquad (1)$$

where functional symbol $G : X \to Y$ stays for the objective function, predicate symbol $p \subseteq X$ stays for (i.e. represents or is interpreted by) a known predicate,

functional symbol $f : X \to Y$ stays for a known function, functional symbol $g : X^* \to X$ stays a known function with a variable (but finite) number of arguments $n(x)$, and all functional symbols $t_i : X \to X$, $i \in [1..n(x)]$ stays for known functions also.

It is well-known [8] that the recursive program scheme (with *uninterpreted* symbols) *is not equivalent* to any *standard program scheme* (i.e. a flowchart with uninterpreted symbols with fixed amount of memory), but can be translated to a *program scheme with stack*. Let us remark that in this paper we consider the scheme of recursive Dynamic Programming with *interpreted* symbols. Usually this interpretation is explicit. For example, in Dropping Bricks Puzzle we have

$$G(H) \;=\; if\ H = 0\ then\ 0\ else\ (1 + \min_{1 \le h \le H} \max\{(h-1), G(H-h)\}).$$

Let us compute a value of this function G for a particular argument by exercising the above recursive algorithm in the left-recursive order:

$$\begin{aligned}
&G(4) \;=\; 1 + \min_{1 \le h \le 4} \max\{(h-1), G(4-h)\} \;= \\
&=\; 1 + \min\{\max\{0, G(3)\}\ ,\ \max\{1, G(2)\}\ ,\ \max\{2, G(1)\}\ ,\ \max\{3, G(0)\}\} \;= \\
&=\; 1 + \min\{\max\{0, 1 + \min\{\max\{0, G(2)\}\ ,\ \max\{1, G(1)\}\ ,\ \max\{2, G(0)\}\}\}\ , \\
&\qquad\qquad \max\{1, G(2)\}\ ,\ \max\{2, G(1)\}\ ,\ \max\{3, G(0)\}\} \;= \\
&=\; 1 + \min\{\max\{0, 1 + \min\{\max\{0, 1 + \min\{\max\{0, G(1)\}\ ,\ \max\{1, G(0)\}\}\}\}\ , \\
&\qquad\qquad \max\{1, G(1)\}\ ,\ \max\{2, G(0)\}\}\}\ ,\ \max\{1, G(2)\}\ ,\ \max\{2, G(1)\}\ , \\
&\qquad\qquad\qquad\qquad \max\{3, G(0)\}\} \;= \ldots \;=\; 3.
\end{aligned}$$

This exercise illustrates what is called a *descending* Dynamic Programming.

One can remark that in the above example we *recompute* values of G for some arguments several times ($G(2)$ and $G(1)$ in particular). This observation leads to an idea to compute function values for new argument values once, then save them (in a hash-table for example), and use them on demand (i.e. instead re-computation). This technique is known in Functional Programming as *memoization* [4].

Some authors claim that *Recursion + Memoization = Dynamic Programming* [4], but we do not think so due to the following reasons. The first one is historical, since the foundational paper [5] did not discuss memoization at all. The second counterargument relies upon observation that recursion in Dynamic programming has a very special form (never nesting in particular). And finally, there exists also an *iterative form* of Dynamic Programming, that we discuss below.

Definition 2. *Let us consider a function $G : X \to Y$ that is defined by scheme (1) of recursive Dynamic Programming. For every argument value $v \in X$, such that $p(v)$ does not hold, let* base *be the following set $bas(v)$ of values $\{t_i(v) : i \in [1..n]\}$. For every argument value v let* support *be the set $spp(v)$ of all argument values that occur in the computation of $G(v)$.*

Proposition 1. *Let us consider a function $G : X \to Y$ that is defined by interpreted scheme (1) of recursive Dynamic Programming. For every argument value $v \in X$, if the objective function G is defined for v, then $spp(v)$ is finite*

and it is possible to pre-compute *(i.e. compute prior to computation of $G(v)$) the support $spp(v)$ according to the following recursive algorithm*

$$spp(x) \;=\; if\ p(x)\ then\ \{x\}\ else\ \{x\} \cup (\bigcup_{y \in bas(x)} spp(y)). \tag{2}$$

Proof. Correctness of the recursive algorithm 2 can be proved by induction on *recursion depth* in computation of $G(v)$. Ii $G(v)$ is defined then finiteness of $spp(v)$ follows from *König's lemma*, since $bas(u)$ is finite for every argument value u where bas is defined. ■

Definition 3. *Let us consider a function $G : X \to Y$ that is defined by scheme (1) of recursive Dynamic Programming. Let us say that a function $SPP : X \to 2^X$ is an* upper support approximation*, if for every argument value v, the following conditions hold:*

- $v \in SPP(v)$,
- $spp(u) \subseteq SPP(v)$ *for every* $u \in SPP(v)$,
- *if* $spp(v)$ *is finite then* $SPP(v)$ *is finite.*

In the case when support or its upper approximation is easy to compute, it makes sense to use *iterative ascending Dynamic Programming* instead of recursive descending Dynamic Programming with memoization.

Ascending Dynamic Programming comprises the following steps.

1. Input argument value v and compute $SPP(v)$. Then compute and save values of the objective function G for all argument values u that are in $SPP(v)$ such that $p(u)$. For example, in Dropping Bricks Puzzle, if we would like to compute value $G(H)$, then $spp(H) = [0..H]$ and the unique argument value of this kind is 0, and, hence, the unique function value that should be saved is $G(0)$; one can save this value as element $G[0]$ of integer array $G[0..H]$.
2. Expand the set of saved values of the objective function by values that can be immediately computed on base of the set of saved values: for every $u \in SPP(v)$, if $G(u)$ is not computed yet, but for every $w \in bas(u)$ value $G(w)$ has been computed and saved already, then compute and save $G(u) = g(u, (G(t_i(u)), i \in [1..n]))$. For example, in Dropping Bricks Puzzle, if values $G(0)$, ... $G(K)$ have been saved in array $G[0..H]$ in elements $G[0]$, ... $G[K]$ (where $0 \le K < H$), one can compute value $G(K+1) = 1 + \min_{1 \le k leq K} \max\{(k-1), G(H-k)\}$ and save it $G[K+1]$.
3. Repeat step 2 until the moment, when you save the value of the objective function for the argument value v. For example, Dropping Bricks Puzzle, step 2 should be executed H times and terminate after saving $G[H]$ in $G[0..H]$.

Let us observe that the ascending Dynamic Programming has not a recursive form but the iterative one.

3 Computing the Least Fix-Point

Let us formalize iterative ascending Dynamic Programming by means of imperative pseudo-code annotated by precondition and postcondition [6,3], i.e. by triples in the following form $\{B\}A\{C\}$, where A is an algorithm in pseudo-code, B — is a logical precondition, and C — is a logical postcondition. A triple $\{B\}A\{C\}$ is said to be valid (or that the algorithm A is partially correct with respect to precondition B and postcondition C), if every terminating exercise of A for input data that satisfy B, the output data satisfy C.

Formalization of the ascending Dynamic Programming follows.

```
\\Precondition:
{D is a non-empty set of argument values,
S and P are ''trivial'' and ''target'' subsets in D,
F : 2^D →2^D is a call-by-value total monotone function,
ρ : 2^D×2^D →Bool is a call-by-value total function
monotone on the second argument}
\\Template:
var Z:= S, Z1 : subsets of D;
repeat Z1:= Z ; Z:= F(Z) until (ρ(P,Z) or Z=Z1)
\\Postcondition:
{ρ(P,Z) ⇔ ρ(P,T),
where T is the least fix-point of the mapping λQ.(S ∪ F(Q))}
```

We would like to refer this formalization as *ascending Dynamic Programming template*, since (as we will see in the section 4) many particular instances of ascending Dynamic Programming algorithms can be generated from this template by specialization of the domain D, sets S and P, and function F. (Like many instances of backtracking and branch-and-bound algorithms can be generated from the unified template that is presented and verified in [12].)

Partial correctness of the formalized ascending Dynamic Programming template follows from Knaster-Tarski fix-point theorem [9]. We would not like to present the exact formulation of the theorem, but would like to present the following proposition that is a trivial corollary from the theorem.

Proposition 2. *Let D be a non-empty set, $G : 2^D \to 2^D$ — be a total monotone function, and R_0, R_1, ... be the following sequence of D-subsets: $R_0 = \emptyset$ and $R_{k+1} = G(R_k)$ for every $k \geq 0$. Then there exists the least fix-point $T \subseteq D$ of the function G and $R_0 \subseteq R_1 \subseteq R2 \subseteq \ldots R_k \subseteq R_{k+1} \subseteq \ldots \subseteq T$.*

The following proposition is a trivial consequences of the above proposition.

Proposition 3. *Dynamic Programming template is partially correct, i.e. for any input data that meets the precondition, the algorithm instantiated from the template either loops or halts in such a way that the postcondition holds upon the termination. Assuming that for some input data the precondition of the Dynamic Programming template is valid, and the domain D is finite, then the algorithm instantiated from the template terminates on these data after (at most) $|D|$ iterations of the loop* `repeat-until`.

Proof. Let us assume that a particular instance of the template terminates for some input data that meets the precondition. According to the above proposition 2, the following function $G = \lambda Q.(S \cup F(Q)) : \ 2^D \to 2^D$ (that maps every $Q \subseteq D$ to $S \cup F(Q)$) has the least fix-point. Let $R_0 = \emptyset$ and $R_{k+1} = G(R_k)$ for every $k \geq 0$; then for every $k > 0$ values of set variables `Z` and `Z1` immediately after k iterations of the loop are R_{k+1}, and R_k respectively, and (according to proposition 2) $R_k \subseteq T$, where is the least fix-point of the mapping G. Hence, if the repeat-loop terminates due to condition $\rho(P, \mathrm{Z})$, then $\rho(P, T)$ due to monotonicity of G; if this loop terminates, but not due to the condition $\rho(P, \mathrm{Z})$ (i.e. this condition is not valid), then it terminates due to another condition `Z=Z1`, that implies that the final value of `X` is equal to the least fix-point T, and hence $\rho(P, T)$ is not valid also. ■

4 Examples of the Template Specialization

In this section we illustrate how the ascending Dynamic Programming template works, i.e. how concrete algorithms can be generated from it by specialization (i.e. by instantiating concrete functions and predicates).

4.1 Computing Dynamic Programming

Let us start with Dropping Bricks Puzzle and adopt

- D to be an "initial segment" of the graph of the function G, i.e. the set of all integer pairs $(m, G(m))$, where m represents a level (in $[1..H]$);
- S to be a singleton set $\{(0, 0)\}$ that consists of the unique trivial pair, and P to be another singleton set $\{(H, G(H))\}$;
- F to be a function that maps any $Q \subseteq D$ to $\{(m, n) \in D \mid$ there exist integers $n_0, \ldots n_{m-1}$ such that $(0, n_0), \ldots (m-1, n_{m-1}) \in Q$ and $n = 1 + \min_{1 \leq k \leq m} \max\{(k-1), n_{m-k}\}\}$;
- $\rho(P, Q)$ to be $\exists n : (H, n) \in (P \cap Q)$.

This specialization meets the precondition of the template of the ascending Dynamic Programming, and D is the least fix-point of F. Hence (according to proposition 3), the resulting algorithm terminates after H iterations of the repeat-loop (since $|D| = H$), and (upon the termination) $(H, G(H)) \in$`Z` (since $\exists n : (H, n) \in (P \cap$`Z`$) \Leftrightarrow \exists n : (H, n) \in (P \cap T)$ where T is D, the fix-point of F), but there is no any other $n \neq G(H)$ such that $(H, n) \in$`Z` (since P is a singleton).

The above example can be generalized as follows.

Proposition 4. *Let us consider a function $G : X \to Y$ that is defined by scheme (1) of recursive Dynamic Programming. Assume that $SPP : X \to 2^X$ is some upper approximation of the support function for G. Let $v \in X$ be any value. If to adopt*

- *the graph of G restricted on $SPP(v)$ as D,*
- *a set $\{(u, f(u)) | p(u)\ \&\ u \in SPP(v)\}$ as S,*

- *a singleton* $\{(v, G(v))\}$ *as* P,
- *a mapping* $Q \mapsto \{(u, w) \in D \mid \exists w_1, \ldots w_n : (t_1(u), w_1), \ldots (t_n(u), w_n) \in Q \ \&\ w = g(u, w_1, \ldots w_n)\}$ *as* $F : 2^D \to 2^D$,
- $\exists w : (v, w) \in (R \cap Q)$ *as* $\rho(R, Q) : 2^D \times 2^D \to Bool$,

then the algorithm that results from the template of the ascending Dynamic Programming computes $G(v)$ *in the following sense: it terminates after iterating repeat-loop* $|SPP(v)|$ *times at most, upon the termination* $(v, G(v)) \in \mathtt{Z}$ *and there is no any* $w \in Y$ *(other than* $G(v)$*) such that* $(v, w) \in \mathtt{Z}$.

Proof. The described specialization meets the precondition of the template of the ascending Dynamic Programming, and D is the least fix-point of F. Hence (according to proposition 3), the resulting algorithm terminates after at most $|SPP(v)|$ iterations of the repeat-loop (since $|D| \leq |SPP(v)|$), and (upon the termination) $(v, G(v)) \in \mathtt{Z}$ (since $\exists w : (v, w) \in (P \cap \mathtt{Z}) \Leftrightarrow \exists w : (v, w) \in (P \cap T)$ where T is D, the fix-point of F), but there is no any other $w \neq G(v)$ such that $(v, w) \in \mathtt{Z}$ (since P is a singleton). ■

4.2 Context-Free Parsing

Parsing theory for context-free (C-F) languages is well established and developed technology [1,2]. The first sound and efficient algorithm for parsing C-F languages was developed independently by J. Cocke, D.H. Younger and T. Kasami in period from 1965 to 1970. More efficient and practical parsing algorithms have appeared since these times, nevertheless Cocke - Younger - Kasami algorithm (CYK algorithm) still has educational importance nowadays[2]. A context-free grammar (C-F grammar) is a tuple $G = (N, E, P, S)$, where

- N and E are disjoint finite alphabets of *non-terminals* and *terminals*,
- $P \subseteq N \times (N \cup E)^*$ is a set of *productions* of the following form $n \to w$, $n \in N$, $w \in (N \cup E)^*$,
- $s \in N$ is the *initial* non-terminal.

A C-F grammar is in the Chomsky Normal Form (CNF) if the initial symbol does not occur in the right-hand side of any production, and every production has the form $n \to n'n''$ or $n \to e$, where $n, n', n'' \in N$ and $e \in E$. *Derivation* in a C-F grammar G is a finite sequence of words $w_0, \ldots w_k, w_{k+1}, \ldots w_m \in (N \cup E)^*$, $(m \geq 0)$, such that every word w_{k+1} within this sequence results from the previous one w_k by applying a production (in this grammar). For every words $w', w'' \in (N \cup E)^*$ let us write $w' \Rightarrow w''$ if there exists a derivation that starts with w' and finishes with w''. Language $L(G)$ generated by the grammar G is defined as follows: $L(G) = \{w \in E^* \mid s \Rightarrow w\}$.

Two C-F grammars are said to be equivalent if they generate equal languages. It is well-known fact that every C-F grammar that does not generate the empty word is equivalent to some CNF grammar [1].

[2] Recently M. Lange and H.F. Leiß suggested a generalized CYK algorithm for educational purposes [10].

Definition 4. *Assume that $G = (N, E, P, s)$ is a given C-F grammar.* Parsing problem *for $L(G)$ can be formulated as follows: for input word $w \in E^*$ construct the set of all pairs (n, u), $n \in N$ and $u \in E^*$ is a (non-empty) subword of w, such that $n \Rightarrow u$.*

In the sequel we discuss parsing problem for CNF grammars only. Let $G = (N, E, P, s)$ be a CNF grammar, $w \in E^*$ be the input word, $L = L(G)$ be the corresponding language, D be the set of all pairs (n, u), where $n \in N$ and $u \in E^*$ is a (non-empty) subword of w, and $T = \{(n, u) \in D \mid n \Rightarrow u\}$. It is easy to see that T is the least fix-point of the following monotone function $F : 2^D \to 2^D$ that maps every $Q \subseteq D$ to $F(Q) = \{(n, e) \in D \mid e \in E,\ (n \to e) \in P\} \cup \{(n, u) \in D \mid \exists (n^p rime, u^p rime), (n^p rime^p rime, u^p rime^p rime) \in Q : u \equiv u^p rime u^p rime^p rime$ and $(n \to n^p rime n^p rime^p rime) \in P\}$. Hence we can adopt $\{(n, e) \in D \mid e \in E,\ (n \to e) \in P\}$ as S, $\{(s, w)\}$ as P in the ascending Dynamic Programming template, and predicate $FALSE$ as ρ. After this specialization the template becomes CYK algorithm that solves the parsing problem for $L(G)$ by iterating repeat-loop at most $|N| \times |w|$ times.

5 Inverting Descending Dynamic Programming

Definition 5. *Let $G : X \to Y$ be a function. A function $G^- : Y \to X$ is said to be* inverse *of G if the following properties hold:*

- *for every $w \in Y$, if $w \in G(X)$ then $G^-(w)$ is defined and $G(G^-(w)) = w$;*
- *for every $w \in Y$, if $w \notin G(X)$ then $G^-(w)$ is undefined.*

Let us remark, that if a function $G : X \to Y$ is not injective, then G^- is not unique.

Let us assume that some total function $G : X \to Y$ is defined by recursive scheme of Dynamic Programming 1. Let us assume also that X is countable (with some fixed enumeration $cnt : N \to X$), that we have the following abstract data type $SubSet$ which values are subsets of X (i.e. all subsets, not just finite), that has standard set-theoretic operations *union* and *intersection* (applicable when *at least one* argument is finite) and another choice operation $fir : SubSet \to X$ that computes for every set T the element of T with *the smallest number* (according to cnt).

Assume that we want to design an algorithm that computes *some* inverse of G. The simplest way to compute $G^-(w)$ for a given $y \in Y$ is to proceed one by one according to count as follows.

\\ `Precondition:`
$\{G : X \to Y$ `is a total computable function,`
X `is a countable set,`
$Fir : SubSet \to X$ `is a choice function,` $y \in Y\}$
\\ `Algorithm:`

```
var x: X; var z: Y; var R:=X: SubSet;
repeat x:= Fir(R); z:= G(x); R:= R\{x} until (z=y or R=∅);
```

if y≠z then loop
\\ Postcondition:
$\{G(x) = y\}$.

Partial correctness of this algorithm is straightforward, but without memoization this algorithm is extremely inefficient. More efficient way to compute the inverse function is presented below as the following *Inverse Dynamic Programming algorithm.*

\\ Precondition:
$\{G : X \to Y$ is a computable function
defined by scheme of recursive Dynamic Programming (1),
$SPP : X \to 2^X$ is an upper support approximation for G,
X is a countable set, $Fir : SubSet \to X$ is a choice function,
$y \in Y\}$
\\ Algorithm:
var x: X; var R:=X, T: SubSet;
var D:=$\emptyset$: $2^{X \times Y}$;
var k : integer;
repeat x:= Fir(R); T:= SPP(x); R:= R\T;
D:= D $\cup$ $\{(u, f(u)) \mid p(u)\ \&\ u \in T\}$;
exercise k$\in [1..|T|]$ times: D:= D $\cup$ $\{(u, w) \notin D \mid$
$\exists w_1, \ldots w_n :\ (t_1(u), w_1), \ldots (t_n(u), w_n) \in D,$
$t_1(u), \ldots t_n(u) \in T,\ \&\ w = g(u, w_1, \ldots w_n)\}$
until $(\exists u : (u, y) \in D$ or R=$\emptyset$);
if $\exists u : (u, y) \in D$ then x:= (u such that $(u, y) \in D$) else loop
\\ Postcondition:
$\{G(x) = y\}$. (Parameter k may be any in the specified range and, maybe, it can be determined by supercompilation [13,14].)

Proposition 5. *Inverse Dynamic Programming algorithm is partially correct.*

Proof. One can proceed according to Floyd - Hoare method [6,3] and use the following (one and the same) invariant to both loops (i.e. for the external repeat-loop and for the internal exercise-loop): D is a subset of graph of G. ■

Proposition 6. *Assume that for some input data the precondition of the Inverse Dynamic Programming algorithm is valid and that the input value y belongs to $G(X)$. Then the algorithm eventually terminates.*

Proof. A standard way to prove algorithm (and program) termination is via *potential* (or bound) function [6,3], i.e. a function that maps states of the algorithm to natural numbers so that every legal loop execution reduces value of the function. Let $n \in N$ be an integer such that $y = G(cnt(n))$, let $m = \sum_{0 \le i \le n} |SPP(cnt(i))|$ and let $\pi(D) = m - |D|$. Then every legal iteration of any loop of our algorithm reduces the value of this function $\pi(D)$ at least by one. ■

As follows from propositions 5 and 6, the inverse Dynamic Programming really computes an inverse function for a function defined by recursive scheme for descending Dynamic Programming.

Let us present an example. It does not make sense to invert function G that solves Dropping Bricks Puzzle, since this function is not injective. So let us consider a simpler injection function $F : N \to N$

$$F(n) \;=\; if\ (n = 0\ or\ n = 1)\ then\ 1\ else\ F(n-1) + F(n-2)$$

that computes Fibonacci numbers. Let us assume that *cnt* is enumeration in the standard order. Then our Inverse Dynamic Programming algorithm gets the following form:

```
var x: N; var R:=N, T: 2^N;
var D:=∅ : 2^(N×N);
var k : integer;
repeat x:= Fir(R); T:= [0..x]; R:= R\[0..x];
     D:= D ∪ {(0,1) | 0 ∈[0..x]} ∪ {(1,1) | 1 ∈[0..x]};
     exercise k∈[1..x] times: D:= D∪ {(u,w) ∉ D |
                                        ∃w1,w2 : (u−1,w1),(u−2,w2) ∈ D,
                                  (u−1),(u−2) ∈[0..x], & w = w1 + w2}
until ∃u : (u,y) ∈ D;
if ∃u : (u,y) ∈ D then x:= (u such that (u,y) ∈ D) else loop.
```

After some simplification one can get the following algorithm: `var x: N; var T: 2^N; var D:=∅: 2^(N×N); var k : integer;`

```
x:=0; D:= {(0,1),(1,1)};
repeat x:=x+1; D:= D ∪ {(x,w1 + w2) |
                                        ∃w1,w2 : (x−1,w1),(x−2,w2) ∈ D};
until ∃u : (u,y) ∈ D;
if ∃u : (u,y) ∈ D then x:= (u such that (u,y) ∈ D) else loop
```

that just computes and saves Fibonacci sequence in the "array" `D` and checks whether y is in the array already.

6 Concluding Remarks

Author would not like to forth everyone to think about Dynamic Programming in terms of fix-point computations, but believe that ascending Dynamic Programming template presented in the paper will help to teach and (maybe) automatize Algorithm Design. This approach to teaching Dynamic Programming is in use in Master program at Information Technology Department of Novosibirsk State University since 2003. A possible application of the unified template is data-flow parallel implementation of the Dynamic Programming, but this topic need more research.

Acknowledgments. The research is supported by joint Russian-Korea project RFBR-12-07-91701-NIF-a.

References

1. Aho, A.V., Ullman, J.D.: The Theory of Parsing, Translation, and Compiling, Vol. 1, Parsing. Prentice Hall (1972)

2. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools, 2nd Edition. Addison-Wesley (2007)
3. Apt, K.R., de Boer, F.S., Olderog, E.-R.: Verifcation of Sequential and Concurrent Programs. Third edition, Springer (2009)
4. Astapov, D.: Recursion + Memoization = Dynamic Programming. (In Russian.) Practice of Functional Programming n.3,17–33, available at `http://fprog.ru/2009/issue3/` (2009)
5. Bellman, R.: The theory of dynamic programming. Bulletin of the American Mathematical Society 60, 503–516 (1954)
6. Gries, D.: The Science of Programming. Springer (1987)
7. Greibach, S.A.: Theory of Program Structures: Schemes, Semantics, Verification. Lecture Notes in Computer Science 36, Springer, Heidelberg (1975)
8. Kotov V.E., Sabelfeld V.K.: Theory of Program Schemata. (Teoria Skhem Programm.) Science (Nauka), Moscow (1991) (in Russian)
9. Knaster, B., Tarski, A.: Un theoreme sur les fonctions d'ensembles. Ann. Soc. Polon. Math., 6, 133–134 (1928)
10. Lange, M., Leiß, H.: To CNF or not to CNF? An Efficient Yet Presentable Version of the CYK Algorithm. Informatica Didactica 8, available at `http://www.informatica-didactica.de/cmsmadesimple/index.php?page=LangeLeiss2009_en` (2009).
11. Shilov, N.V.: A note on three Programming Paradigms. In: 2nd International Valentin Turchin Memorial Workshop on Metacomputation in Russia, pp.173–184. Ailamazyan Program Systems Institute, Pereslavl-Zalessky, Russia (2010)
12. Shilov, N.V.: Algorithm Design Template base on Temporal ADT. Proceedings of 18th International Symposium on Temporal Representation and Reasoning, IEEE Computer Society, 157–162 (2011)
13. Turchin, V.F.: The concept of a supercompiler. ACM Transactions on Programming Languages and Systems, 8(3), 292–325 (1986)
14. Turchin, V.F.: Supercompilation: the approach and results. (Superkompilyatsya: metody i rezultaty.) In: Current trends in architecture, design and implementation of program systems. (Problemy arkhitektury, analiza i razrabotki programmnyh system.). System Informatics (Sistemtaya Informatika) 6, Science (Nauka), Novosibirsk, 64–89 (1998) (in Russian)