Bull. Nov. Comp. Center, Comp. Science, 34 (2012), 135–155

Unifying dynamic programming design patterns

N. V. Shilov

Abstract. Design and Analysis of Computer Algorithms is a must of Computer Curricula. It covers many topics that group around several core themes including algorithmic design patterns (ADP) like the greedy method, divide-and-conquer, dynamic programming, backtracking and branch-and-bound. These design patterns are usually considered as Classics of the Past (going back to days of R. Floyd and E. Dijkstra). However, ADP can be (semi)formalized as design templates, specified by correctness conditions, and formally verified either in the Floyd-Hoare methodology, by means of the Manna-Pnueli proof-principles, or in some other way. This approach has lead to new insights and better comprehension of the design patterns, of specification and verification methods. Formalization of backtracking (BT) and branch-and-bound (B&B) ADP has been presented at TIME-2011 symposium. In the present paper, we suggest and discuss a formalization of Dynamic Programming. A methodological novelty consists in treatment (interpretation) of ascending Dynamic Programming as least fix-point computation (according to the 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. (Examples of the latter are the Cocke-Younger-Kasami parsing algorithm and computation of inverse for a total function.) This interpretation leads also to an opportunity to design, specify and verify (1) a unified template for imperative Dynamic Programming, (2) a unified template for inverting Dynamic Programming (in countable domains), and may lead to (3) a unified template for data-flow implementation of Dynamic Programming. The present paper is a revised and extended version of the original publication in the Proceedings of the 3rd Workshop on Metacomputation (2012) that focused on inverting Dynamic Programming.

1. Introduction

1.1. Dropping bricks from a high tower

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

> Let us characterize the mechanical stability (strength) of a brick by an integer h that is equal to the height (in meters) safe for the brick to fall down, while height $(h+1)$ meters is unsafe (i.e.

[1]When this paper had been drafted, Prof. Teodor Zarkua (St. Andrew University of Georgian Patriarch) informed the author that the problem is known already and had been used for programming contests (check, for example, the problem at URL `http://acm.timus.ru/problem.aspx?space=1&num=1223`). Some time ago 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`).

> the brick breaks). You have to determine the 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 a safe fall.) 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 will sketch a descending Dynamic Programming solution of the above problem as a gentle introduction to Dynamic Programming approach to optimization, design its implementation in terms of the functional pseudo-code and give some historical remarks. At the end of this section we are going to introduce what we call a *scheme of recursive descending Dynamic Programming* and discuss in brief how to improve the efficiency of recursive Dynamic Programming by *memoization*.

The rest of the paper is organized as follows. In Section 2, we will translate recursive Dynamic Programming into the *iterative* form that corresponds to the *ascending Dynamic Programming*. This translation is based on the interpretation of the ascending Dynamic Programming as computations of *the least fix-point* of a 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 applications/specializations (solving finite games and context-free parsing) are presented in Section 3. Problems with translation of recursive descending Dynamic Programming to iterative ascending Dynamic Programming are discussed in Section 4: in particular, we prove in this section that in the general case of Dynamic Programming static memory is not enough and dynamic memory is needed, and discuss backtracking vs. Dynamic Programming (by a study of the discrete knapsack problem). In Section 5, we will suggest an approach of inverting total functions defined by descending Dynamic Programming in countable domains. In the last Section we conclude our paper with discussing topics for further research and, in particular, the *data-flow* approach to ascending Dynamic Programming.

The present paper is a revised and extended version of the original publication in the Proceedings of the 3rd International Valentin Turchin Workshop on Metacomputation (2012) [18]. That preliminary publication focused on the problem of inverting recursive descending Dynamic Programming and contained some minor errors that have been corrected in the present paper.

1.2. Recursive method for optimization problems

The Dropping Bricks Puzzle is a particular and explicit example of optimization problems. Originally, Dynamic Programming was designed as a

recursive *search* (or construction) of an optimal *program* (or plan) that remains optimal at every stage. In particular, let us consider the puzzle below.

Any optimal method of defining the 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 holds for this particular level 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 the 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 the 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*, the above equality transforms to the following 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 the sequence of integers G_0, G_1, ... G_H, ... that meets these two equalities is unique since G_0 is defined explicitly, G_1 is defined by G_0, G_2 is defined by G_0 and G_1, G_H is defined 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[2] $G : \mathbb{N} \to \mathbb{N}$ that maps every natural H to G_H and satisfies the following *functional equation* for the *objective function* G:

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

This equation has a unique solution as it follows from the uniqueness of the sequence G_0, G_1, ... G_H, ... Hence it can be adopted as a recursive definition of a function, i.e. a recursive algorithm presented in a functional pseudo-code. This is an example of the historically first 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 the 1950s [6]. At this time, the

[2] $\mathbb{N}$ is the set of of natural numbers $\{0, 1, 2, \dots\}$.

noun *programming* had nothing in common with more recent *computer programming* and meant *planning* (compare: *linear programming*). The adjective *dynamic* points out that Dynamic Programming is related to a *change of state* (compare: *dynamic logic*, *dynamic system*). Functional equations for the objective function similar to the above are called *Bellman equations*. They formalize the following *Bellman Principle of Optimality*, which we have used implicitly in the puzzle: an optimal program (or plan) remains optimal at every stage.

At the same time, according to [8], R. Bellman, speaking about the 1950s, explains:

> An interesting question is, "Where did the name, *dynamic programming*, come from?" The 1950s were not good years for mathematical research. (...) Hence, I felt I had to do something to shield [the Secretary of Defense] and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation. (...) Let's take a word that has an absolutely precise meaning, namely dynamic, in the classical physical sense. It also has a very interesting property as an adjective, and that is it's impossible to use the word dynamic in a pejorative sense. Try thinking of some combination that will possibly give it a pejorative meaning. It's impossible. I thought dynamic programming was a good name. It was something not even a Congressman could object to. So I used it as an umbrella for my activities.

1.3. From recursion to iterative Dynamic Programming

If we analyze the recursive Dynamic Programming methodology accumulated in the Bellman Principle (in particular, in the above recursive solution for the Dropping Bricks Puzzle), it is possible to suggest the following *scheme of recursive (descending) Dynamic Programming*:

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

where function $G : X \to Y$ is the objective function, $p \subseteq X$ is a known predicate, $f : X \to Y$ is a known function, $g : X^* \to X$ is a known function with a variable (but finite) number of arguments to be defined by a known function $n(x) : X \to \mathbb{N}$, and all $h_i : Y \to X$, $t_i : X \to X$, $i \in [1..n(x)]$ stays for known functions as well. In principle, here we understand the scheme of recursive Dynamic Programming in the sense of the *theory of program schemata* [10, 11], i.e. we assume that p, f, g, n, all h_i and g_i are *uninterrupted* predicate and functional symbols that have to be *interpreted* to define a functional program and/or a Bellman equation for a concrete problem. From here on, we will make this difference implicit rather than

implicit. In particular, for the Dropping Bricks Puzzle $G(x) = if\ x = 0\ then\ 0\ else\ (1 + \min 1_{1 \leq i \leq x} \max\{(i-1), G(x-i)\})$, we have

1. predicate $\lambda x.(x = 0)$ is interpretation for p,
2. constant function $\lambda x.0$ is interpretation for f,
3. identical function $\lambda x.x$ is interpretation for n,
4. for every $i \in [1..n(x)]$, function $\lambda x.(x - i)$ is interpretation for t_i,
5. for every $i \in [1..n(x)]$, function $\lambda t.\max\{(i-1), t\}$ is interpretation for h_i,
6. function $\lambda x.\lambda w_1 \ldots \lambda w_n.(\min_{1 \leq i \leq x} w_i)$ is interpretation for g.

Let us compute the value of this function G for a particular argument by exercising the above recursive algorithm in the left-recursive order:
$G(4) = 1 + \min_{1 \leq h \leq 4} \max\{(h-1), G(4-h)\} =$
$= 1+\min\{\max\{0, G(3)\}\ ,\ \max\{1, G(2)\}\ ,\ \max\{2, G(1)\}\ ,\ \max\{3, G(0)\}\} =$
$= 1+\min\{\max\{0, 1+\min\{\max\{0, G(2)\}\ ,\ \max\{1, G(1)\}\ ,\ \max\{2, G(0)\}\}\}\ ,$
$\max\{1, G(2)\}\ ,\ \max\{2, G(1)\}\ ,\ \max\{3, G(0)\}\} =$
$= 1+\min\{\max\{0, 1+\min\{\max\{0, 1+\min\{\max\{0, G(1)\}\ ,\ \max\{1, G(0)\}\}\}\ ,$
$\max\{1, G(1)\}\ ,\ \max\{2, G(0)\}\}\}\ ,\ \max\{1, G(2)\}\ ,\ \max\{2, G(1)\}\ ,$
$\max\{3, G(0)\}\} = \ldots = 3.$

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[3] once, then save them , and instead of re-computation use them on demand. This technique is known in Functional Programming as *memoization* [5]. One can distinguish *functional* and *imperative* styled memoization. Functional memoization consists in extending the clause base by a new clause $G(i) = j$ as soon $G(i)$ is computed and $G(i) = j$. Imperative memoization consists in saving computed values in a hash-table, for example.

Some authors claim that *Recursion + Memoization = Dynamic Programming* [5]. We do not share this view for the following reasons. The first one is that the foundational paper [6] made no mention of memoization. The second counterargument relies upon the observation that recursion in Dynamic programming has a very special form (in particular, it never nests). And finally, there exists also an *iterative form* of Dynamic Programming to be discussed below. This form of Dynamic Programming does not rely upon memoization but precomputes some values in advance.

[3] i.e. for argument values that have never occurred before

2. Ascending Dynamic Programming template

2.1. Informal discussion

Let us consider a function $G : X \to Y$ that is defined by the 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 $\{t_i(v) : i \in [1..n(v)]\}$. Let us remark that for every argument value v, if $G(v)$ is defined, $bas(v)$ is finite. Let us also observe that if the objective function G is defined for some argument value v, then it is possible to *precompute* (i.e. compute prior to the computation of $G(v)$) the *support* for this argument value v, i.e. the set $spp(v)$ of all argument values that occur in the recursive computation of $G(v)$ according to the following recursive algorithm

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

Another remark is that for every argument value v, if $G(v)$ is defined, then $spp(v)$ is finite (since computation of $G(v)$ terminates). 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.

Let us consider the case when the support function or its upper approximation is *easier to compute*, i.e. the (time and/or space) complexity of the available algorithm to compute the support function or its upper approximation is better than the complexity of the available algorithm that computes G. Then 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 $SPP(v)$. Let G be an array[4] of Y indexed by values in $SPP(v)$. Then compute and save values of the objective function G for all values $u \in SPP(v)$ such that $p(u)$: $G[u] := f(u)$.

    - For example, in the Dropping Bricks Puzzle, if we wish to compute the value $G(H)$, then $spp(H) = [0..H]$ and 0 is the unique value $u \in spp(H)$ such that $p(u)$, and the unique function value that should be saved is $G(0)$; save this value in the element $G[0]$ of the integer array $G[0..H]$.

[4]That is (in Pascal style) *var* G : Y *array of* $SPP(v)$.

2. Expand the set of saved values of the objective function by values that can be immediately computed on the basis of the set of saved values: for every $u \in SPP(v)$, if $G(u)$ has not been computed yet, but for every $w \in bas(u)$ the value $G(w)$ has already been computed and saved in $G[w]$, then compute and save $G(u)$ in $G[u]$: $G[u] := g(u,\ (h_i(G(t_i(u))),\ i \in [1..n(u)]))$.

    - For example, in the Dropping Bricks Puzzle, if values $G(0)$, ... $G(K)$ have been saved in the array $G[0..H]$ in elements $G[0]$, ... $G[K]$, $0 \leq K < H$, one can compute the value $G(K+1)$ and save it: $G[K+1] :=\ 1 + \min_{1leqk \leq K} \max\{(k-1),\ G[K-k]\}$.

3. Repeat Step 2 until the moment when the value of the objective function for the argument v is saved.

    - For example, for the Dropping Bricks Puzzle, Step 2 should be executed H times and terminated after saving $G[H]$.

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

2.2. Formalization

Let us formalize iterative ascending Dynamic Programming by means of an imperative pseudo-code annotated by precondition and postcondition [9, 4, 17], i.e. by triples in the form $\{B\}A\{C\}$, where A is a pseudo-code of some algorithm, 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 outputs data that satisfy C.

Formalization of the ascending Dynamic Programming follows.
\\Precondition:
{D is a non-empty set[5] of argument values,
2^D is the corresponding powerset with the standard partial order $\subseteq$,
S and P are "trivial" and "target" subsets in D,
$F : 2^D \to 2^D$ is a call-by-value total monotone function,
$\rho : 2^D \times 2^D \to Bool$ is a call-by-value total function monotone on the second argument}
\\Peseudo-code:
$var\ U := S,\ V\ :\ subsets\ of\ D;$
$repeat\ V := U;\ U := F(V) \cup S\ until\ (\rho(P,U)\ or\ U = V)$

[5]Note that we do not require any *explicit* representation for D; it is just assumed to be a set (a *virtual* set).

\\Postcondition:
$\{\rho(P,U) \Leftrightarrow \rho(P,T)$,
where T is the least subset of D such that $T = (F(T) \cup S)\}$

Here the first (initiating) assignment $U := S$ corresponds to the first step of the informal description of ascending Dynamic Programming, the third assignment $U := F(V)$ in the loop body corresponds to the second step, and loop condition $\rho(P,U)$ corresponds to the condition at the third step; an auxiliary variable V, the second assignment $V := U$ and condition $U = V$ are used for termination in case no further progress is possible.

We would like to refer to this formalization as the *iterative ascending Dynamic Programming template*, since (as we will see in the section 3) many particular instances of ascending Dynamic Programming algorithms can be generated from this template by specialization of the domain D, sets S and P, function F and criterion ρ. (In the same way as many instances of backtracking and branch-and-bound algorithms can be generated from the unified template presented and verified in [16].)

Partial correctness of the formalized ascending Dynamic Programming template follows from the Knaster-Tarski fix-point theorem [12]. Instead of presenting an exact formulation of the theorem we would like to present the following corollary.

Corollary 1. *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 $R_0 \subseteq R_1 \subseteq R2 \subseteq \ldots R_k \subseteq R_{k+1} \subseteq \cdots \subseteq T$.*

The following Propositions 1 and 2 are trivial consequences of the above Corollary.

Proposition 1. *Iterative ascending Dynamic Programming template is partially correct*

Proof. Let us assume that a particular instance of the template terminates for some input data meeting the precondition. According to the above Corollary, the value of T is the least fix-point of the following monotone function $G \equiv \lambda Q.(S \cup F(Q))$ (that maps every $Q \subseteq D$ to $S \cup F(Q)$). 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 U and V immediately after k iterations of the loop are R_{k+1} and R_k, respectively, and, according to the Corollary, $R_k \subseteq T$. Hence, if the *repeat*-loop terminates due to the condition $\rho(P,U)$, then $\rho(P,T)$ due to the monotonicity of ρ. If this loop terminates not due to the condition $\rho(P,U)$ (i.e. this condition is not valid), then it terminates due to another condition

$U = V$ implying that the final value of U is equal to the least fix-point T, and hence $\rho(P, T)$ is not valid. ■

Proposition 2. *Assume that for some input data the precondition of the iterative ascending Dynamic Programming template is valid and the domain D is finite. Then the algorithm generated from the template terminates after $|D|$ iterations of the loop.*

Proof. is straightforward. ■

3. Examples of ascending Dynamic Programming

3.1. Computing functions

Let us start with an application of the template and Propositions 1 and 2 to the Dropping Bricks Puzzle. Let

- D be an "initial segment" of the graph[6] of the function G, i.e. the set of all integer pairs $(m, G(m))$, where m represents a level (in $[1..H]$);
- S be a singleton set $\{(0, 0)\}$ that consists of the unique trivial pair, and P be another singleton set $\{(H, G(H))\}$;
- F be $\lambda Q \subseteq D.\ \{(m, n) \in D \mid$ there exist integers $n_0, \ldots n_{m-1}$ such that $(0, n_0), \ldots (m-1, n_{m-1}) \in Q$ and $n = 1 + \min_{1 \le k \le m} \max\{(k-1), n_{m-k}\}\}$;
- $\rho(P, Q)$ be $\lambda P, Q.(P \subseteq Q)$.

One can object that in the above settings the loop condition $P \subseteq U$ can not be checked, because we do not know $G(H)$ in advance. However, according to definition of D as the initial segment of the graph of the function G, this condition is equivalent to another one $\exists n : (H, n) \in U$ that can be checked. It easy to see that this specialization meets the precondition of the template of the ascending Dynamic Programming and the domain D is finite. Hence the algorithm resulting from this specialization is correct and terminates according to Propositions 1 and 2; the final value of U includes $(H, G(H))$ since in this case the least fixpoint of $F \cup S$ is D.

The above discussion around the Dropping Bricks Puzzle can be generalized for an arbitrary function G defined by recursive scheme 1. Let

- D be $\{(u, G(u)) :\ u \in SPP(v)\}$, where $SPP(v)$ is an upper support approximation;

[6]Remember that the *graph* of a function $G : X \to Y$ is the following set $\{(x, G(x)) : x \in X\} \subseteq X \times Y$.

- S be $\{(u, f(u)) : \ p(u)$ *and* $u \in SPP(v)\}$ and P be a singleton $\{(v, G(v))\}$;
- F be $\lambda Q \subseteq D.\ \{(u, w) \in D \mid n = n(u),$
$\exists w_1, \ldots w_n : (t_1(u), w_1), \ldots (t_n(u), w_n) \in Q,$
and $w = g(u, h_1(w_1), \ldots h_n(w_n))\}$;
- ρ be $\lambda P, Q.(P \subseteq Q)$ (that is equivalent to $\lambda P, Q \subseteq D.(\exists w : (v, w) \in Q)$ in D).

Then, again, the algorithm resulting from the template after this specialization is correct and terminating according to Propositions 1 and 2.

3.2. Solving finite position games

The game theory has a variety of formalization game notions [3]; in this paper we consider a particular class of formalized games called *finite positional games* of two players, A (Alice) and B (Bob). A game of this kind is a tuple $G = (P_A, P_B, M_A, M_B, F_A, F_B)$, where

- P_A and P_B are disjoint finite sets of *positions* for Alice and Bob respectively,
- $M_A \subseteq P_A \times (P_A \cup P_B)$ and $M_B \subseteq P_B \times (P_A \cup P_B)$ are admissible *moves* of Alice and Bob respectively,
- $F_A \subseteq (P_A \cup P_B)$ and $F_B \subseteq (P_A \cup P_B)$ are disjoint *winning* positions for Alice and Bob respectively.

Alice is said to be a counterpart for Bob and vice versa. Any game of the above type (hereinafter, the game) can be considered as an oriented labeled graph, where nodes are positions marked obligatorily either by P_A or P_B and optionally by F_A or F_B, and edges are moves marked by M_A or M_B.

A play of a game is any finite or infinite sequence of positions p_0, ... p_k, ..., where every pair $(p_k, p_{(k+1)})$ is a move of Alice or Bob. A complete play of a game is an infinite play without instances of any winning position or a finite play that has the unique instance of a winning position as the last position of the play; Alice/Bob wins a finite complete play if the final position of the play is a winning position of Alice/Bob, respectively. A strategy for a player $I \in \{A, B\}$ is any subset $M' \subseteq M_I$; a strategy M' for a player $I \in \{A, B\}$ is said to be a winning strategy if the player eventually wins every play by applying this strategy, i.e. every finite play where I applies M' can not be prolonged to a complete infinite play or to a finite complete play that is winning for the counterpart. To solve a game G on behalf of (or for) a player $I \in \{A, B\}$ means to compute the set of positions W_I where the player has a winning strategy. From here on we will solve finite position games on behalf of Alice; the case for Bob is similar.

A game $G = (P_A, P_B, M_A, M_B, F_A, F_B)$ can be solved on behalf of Alice on basis of the following observation: W_A is the least sets of positions that satisfy the following equation:

$$W_A = F_A \cup \\ \cup \{p \in P_A \setminus F_B |\ \exists p' :\ (p,p') \in M_A \textit{ and } p' \in W_A\} \cup \\ \cup \{p \in P_B \setminus F_B |\ \forall p' :\ (p,p') \in M_B \textit{ and } p' \in W_A\}.$$

Let us introduce the following function $WinA$ on subsets of $(P_A \cup P_B)$: $WinA$ maps every $Q \subseteq (P_A \cup P_B)$ to

$$\{p \in (P_A \cup P_B) \setminus F_B | \\ \exists p' :\ (p,p') \in M_A \textit{ and } p' \in Q \textit{ or} \\ \forall p' :\ (p,p') \in M_B \textit{ and } p' \in Q\}.$$

Let $D = (P_A \cup P_B)$, $S = F_A$, $P = W_A$, $F = WinA$, ρ be $\lambda P, Q \subseteq D.(P \subseteq Q)$. According to Propositions 1 and 2, the algorithm resulting from the template after this specialization terminates after $|P_A \cup P_B|$ iterations of the $repeat - until$ loop and computes W_A.

3.3. Parsing context-free languages

The parsing theory for context-free (C-F) languages is a well established and developed technology [1, 2, 17]. The first sound and efficient algorithm for parsing C-F languages was developed independently by J. Cocke, D.H. Younger and T. Kasami in the period from 1965 to 1970. Even though more efficient and practical parsing algorithms have appeared since then, the Cocke-Younger-Kasami algorithm (CYK algorithm) has preserved its educational importance to this day[7].

A context-free grammar (C-F grammar) is a tuple $G = (N, E, R, s)$ where

- N and E are disjoint finite alphabets of *non-terminals* and *terminals*,
- $R \subseteq N \times (N \cup E)^*$ is a set of *productions* (or rules) 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,... w_k, $w_{(k+1)}$, ... w_m in $(N \cup E)^*$, $m \geq 0$ such that every word w_{k+1} within this sequence results from the previous w_k by applying a production (in this grammar). For any pair of 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''. The

[7]Recently M. Lange and H.F. Leiß suggested a generalized CYK algorithm [13].

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 that every C-F grammar that does not generate the empty word is equivalent to some CNF grammar [1].

Assume that $G = (N, E, P, s)$ is a given C-F grammar. The *parsing problem* for $L(G)$ can be formulated as follows: for the input word $w \in E^*$ construct the set of all pairs (n, u) where $n \in N$ and $u \in E^*$ is a non-empty subword of w such that $n \Rightarrow u$. From here on, we are discussing the the 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 $SA = \{(n, u) \in D \mid n \Rightarrow u\}$. It is easy to see that SA is the least subset of D such that

$$SA = \{(n, e) \in D \mid (n \to e) \in R\} \cup$$
$$\cup \{(n, u) \in D \mid \exists (n', u'), (n'', u'') \in SA \ : \ u \equiv u'u'' \text{ and } (n \to n'n'') \in R\}.$$

Let us introduce the following function *derive* on subsets of D: it maps every $Q \subseteq D$ to $\{(n, u) \in D \mid \exists (n', u'), (n'', u'') \in Q : \ u \equiv u'u''$ *and* $(n \to n'n'') \in R\}$. This function is total and monotone on 2^D. Hence we can take $\{(n, e) \in D \mid e \in E, \ (n \to e) \in R\}$ as S, *derive* as F, and $\lambda P, Q \subseteq D.((s, w) \in Q)$ as ρ in the ascending Dynamic Programming template. After this concretization the template becomes CYK algorithm.

4. Limitations of descending dynamic programming

4.1. A need for dynamic memory

We demonstrated in Section 3.1 that every function $G : X \to Y$ defined by the recursive scheme of Dynamic Programming 1 can be computed by an iterative program with a *dynamic array*: for any argument $v \in X$, the desired value $G(v)$ may be computed by an iterative program with the aid of an array with the size $|spp(v)|$. The advantage of the translation is the use of an *array* instead of a *stack* required to translate a general case recursion: an array is much better than a stack, because an array provides memory access in constant time, and a stack provides sequential memory access that requires linear time. Nevertheless the natural question arises: can finite static memory suffice for computing the function G?

Unfortunately, this is not true because of the following Corollary obtained by M.S. Paterson and C.T. Hewitt [14, 11].

Corollary 2. *The recursive scheme*

$$F(x) = \ if\ p(x)\ then\ x\ else\ f(F(g(x)), F(h(x)))$$

is not equivalent to any standard program scheme (i.e. an uninterpreted iterative program scheme with finite static memory).

This statement does not mean that dynamic memory is *always* required; it just means that for *some* interpretations of *uniterpreted* symbols p, f, g and h the size of required memory depends on the input data. But if p, f, g and h are *interpreted*, it may happen that function G can be computed by an iterative program with a finite static memory.

For example, it is possible to prove [19] that for the Dropping Bricks Puzzle

$$G(H) = \min\{n \in \mathbb{N} : \ \frac{n \times (n+1)}{2} \geq H\};$$

hence, the puzzle can be solved by the following simple iterative algorithm with two integer variables:

$$\begin{array}{l} var\ n := 0, H : \ integer; \\ input(H); \\ while\ \frac{n\times(n+1)}{2} < H\ do\ n := n + 1\ od; \\ output(n). \end{array}$$

Two other examples of this kind are the factorial function and Fibonacci numbers

$$Fac(n) = \ if\ n = 0\ then\ 1 else\ n \times Fac(n-1),$$

$$Fib(n) = \ if\ n = 0\ or\ n = 1\ then\ 1\ else\ Fib(n-2) + Fib(n-1).$$

In both cases, three integer variables suffice to compute them:

Factorial function	Fibonacci numbers

var $n,\ f := 1\ :\ integer;$
input$(n);$
while $n > 0$
  do $f := n \times f;$
  $n := n - 1$
  od;
output(f).
	var $n,\ f0 := 1,\ f1 := 1\ :\ integer;$
	input$(n);$
	while $n > 0$
	  do $f1 := f0 + f1;$
	  $f0 := f1 - f0;$
	  $n := n - 1;$
	  od;
	output$(f1)$.

4.2. Backtracking vs. Dynamic Programming

The Knapsack Problem is classics of optimization and algorithms design [7]. The Knapsack Problem can be informally presented as follows:

> Assume that you have a knapsack with the capacity of $W > 0$ kilograms, a set of goods G_1, ... G_n, with weights W_1, ... W_n that cost P_1, ... P_n. You need to collect in the knapsack as much as possible (i.e. the gross weight cannot exceed the capacity W) to maximize the total price of the collection.

The above problem statement can be refined in several different ways. Two extreme formalizations known as the *continuous* and *discrete* knapsack problems are presented below:

- if goods are infinitely dividable, the problem is to compute a real vector $(w_1, \dots w_n) = \ argmax\{\Sigma_{1\le k\le n} P_k \times \frac{w_k}{W_k} \ : \\ 0 \le w_1 \le W_1, \ \dots, \ 0 \le w_n \le W_n \ and \ \Sigma_{1\le k\le n} w_k \ \le W\}$;
- if goods are undividable, the problem is to compute a Boolean vector $(c_1, \dots c_n) = \ argmax\{\Sigma_{1\le k\le n} P_k \times c_k \ : \\ c_1, \ \dots, \ c_n \in \{0,1\} \ and \ \Sigma_{1\le k\le n} P_k \times c_k \ \le W\}$.

Both problems can be formulated à la recursive descending Dynamic Programming, but these formulations are not easy to translate to equivalent iterative programs.

For the sake of simplification, let us consider related problems to compute maximal gross prices that are possible to collect in continuous(C) and in discrete (D) cases:

- $C(W, n) = \ \max\{\Sigma_{1\le k\le n} P_k \times \frac{w_k}{W_k} \ : \\ 0 \le w_1 \le W_1, \dots 0 \le w_n \le W_n \ and \ \Sigma_{1\le k\le n} w_k \ \le W\}$;
- $D(W, n) = \ \max\{\Sigma_{1\le k\le n} P_k \times c_k \ : \\ c_1, \dots c_n \in \{0,1\} \ and \ \Sigma_{1\le k\le n} P_k \times c_k \ \le W\}$.

Corresponding Bellman equations are straightforward:

- $C(z, m) = \ if \ m = 1 \ then \ P_1 \times \frac{\min(z, W_1)}{W_1} \ else \\ \max_{0\le w\le \min\{z, W_m\}} (P_m \times \frac{w}{W_m} \ + \ C((z - w), (m - 1))$;
- $D(z, m) = \ if \ m = 1 \ then \ (if \ W_1 > z \ then \ 0 \ else \ P_1) \ else \\ (if \ W_m > z \ then \ D(z, (m - 1)) \ else \\ \max\{D(z, (m - 1)), \ P_m + D((z - W_m), (m - 1))\})$,

where z is real value in $[0, W]$ and m is integer in $[1..n]$.

However, the first Bellman equation does not match the scheme of recursive Dynamic Programming 1, because $\max_{0\le w\le \min\{z, W_m\}}$ is a function

of an infinite "number" of arguments. The second equation matches the scheme, but in this case it is not easy to compute support according to the standard procedure

$$spp(z,m) = \ if\ m = 1\ then\ \{(z,1)\}\ else\ \{(z,m)\} \cup\ spp(z,(m-1)) \cup \\ \cup\ (if\ W_m > z\ then\ \emptyset\ else\ spp((z-W_m),(m-1))),$$

because the complexity of this computation is $O(2^m)$. So, in this case *backtracking* or *branch and bound* can be more helpful [7, 16] than iterative ascending Dynamic Programming.

Nevertheless, there exists a very important special case of the discrete knapsack problem, when translation to iterative ascending Dynamic Programming can improve efficiency: if it is known that knapsack capacity W as well as weights of all goods are integers (natural numbers), it makes sense to use the following upper approximation $SPP(z,m) = [1..z] \times [1..m]$ instead of $spp(z,m)$. In this case, $D(W,n)$ can be computed (according to guidelines of Section 3.1) as follows:

$var\ n:\ integer;$
$var\ D:\ integer\ array\ of\ [0..W, 0..n];$
$input(n);$
$for\ m := 0\ to\ W\ do\ D[m,0] := 0;$
$for\ m := 1\ to\ W\ do$
  $for\ k := 1\ to\ n\ do$
  $D[m,k] :=\ if\ W_k > m\ then\ D[m,(k-1)]$
    $else\ \max\{D[m,(k-1)],\ (D[(m-W_k),(k-1)] + P_k)\}$
  od
$od;$
$output(D[W,n]).$

Complexity of this algorithm is $O(W \times n)$. Hence this translation can help only if $W \times n$ is less than 2^n; otherwise backtracking or branch and bound, again, can help better.

5. Inverting functions

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

Assume that we wish to design an algorithm that computes *some* inverse of G, i.e. a function $G^{inv} : Y \to X$ having the following properties:

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

Let us remark that the inverse function is unique iff G is injective; otherwise, G has several different inverse functions.

The simplest way to compute $G^{inv}(w)$ for a given $w \in Y$ is the brute-force exhaustive search that proceeds one by one according to *cnt*:
\\ Precondition:
$\{G : X \to Y$ is a total computable function,
X is countable and $Fir : SubSet \to X$ is a choice function, $y \in Y\}$
\\ Pseudo-code:
$let\ G^{inv}(Q : SubSet) =\ if\ (w = G(fir(Q)))$
$then\ fir(Q)$
$else\ G^{inv}(Q \setminus \{fir(Q)\})$
$in\ x := G^{inv}(X)$
\\ Postcondition:
$\{G(x) = y\}$.
Partial correctness of this algorithm is straightforward, as well as termination in the case when $y \in G(X)$. Without memoization, however, this algorithm is extremely inefficient.

A more efficient algorithm can be derived for functions defined by the recursive scheme of Dynamic Programming 1. Since we are looking for *any* $x \in X$ such that $y = G(x)$, we can suggest the following recursive algorithm:
$\{G : X \to Y$ is a total function defined by 1,
$SPP : X \to 2^X$ is an upper support approximation for G, X is countable and $Fir : SubSet \to X$ is a choice function, $y \in Y\}$
let
$let\ z = Fir(Q)$
$in\ G^{inv}(Q : SubSet) =\ if\ \exists u \in SPP(z) : y = G(u)$
$then\ (any\ u \in SPP(z)\ such\ that\ y = G(u))$
$else\ G^{inv}(Q \setminus SPP(z))$
$in\ x := G^{inv}(X)$
$\{G(x) = y\}$.

The above recursive algorithm uses tail recursion; hence it is equivalent to the following iterative one (pre- and post- conditions remain the same):
$var\ x, u, z :\ X;$
$var\ R := X,\ T :\ SubSet;$
$repeat\ z := Fir(R);\ T := SPP(z);\ R := R \setminus T;$
$until\ (\exists u \in T : y = G(u)\ or\ R = \emptyset);$
$if\ (\exists u \in T : y = G(u))\ then\ x :=\ (any\ u \in T\ such\ that\ y = G(u))$
$else\ loop.$

The next step is the use of the ascending iterative Dynamic Programming in validation of the loop condition $\exists u \in T : y = G(u)$. The loop
repeat $z := Fir(R);\ T := SPP(z);\ R := R \setminus T;$
until $(\exists u \in T : y = G(u)$ *or* $R = \emptyset);$
is equivalent to another loop
repeat $z := Fir(R);\ T := SPP(z);\ R := R \setminus T;$
$D := \{(u, f(u))|\ p(u)$ *and* $u \in T\};$
exercise $|T|$ *times* :
$D := D \cup \{(u, w)|\ n = n(u),$
$\exists w_1, \ldots w_n : ((t_1(u), w_1), \ldots (t_n(u), w_n) \in D,$
$w = g(u, h_1(w_1), \ldots h_n(w_n)))\}$
until $(\exists u : (u, y) \in D$ *or* $R = \emptyset),$
because after termination of the internal *exercise*-loop the set variable D contains the graph of G on $SPP(z)$ and is a subset of the graph of G. Hence we get the following algorithm for computing the inversion that we call the *inverse Dynamic Programming template*:
\\ Precondition:
$\{G : X \to Y$ is a total computable function,
X is countable and $Fir : SubSet \to X$ is a choice function, $y \in Y\}$
\\ Pseudo-code:
var $x, u, z :\ X;$
var $R := X,\ T :\ SubSet;$
repeat $z := Fir(R);\ T := SPP(z);\ R := R \setminus T;$
$D := \{(u, f(u))|\ p(u)$ *and* $u \in T\};$
exercise k *times* $(for\ some\ k \in [1..|T|])$:
$D := D \cup \{(u, w)|\ n = n(u),$
$\exists w_1, \ldots w_n : ((t_1(u), w_1), \ldots (t_n(u), w_n) \in D,$
$w = g(u, h_1(w_1), \ldots h_n(w_n)))\}$
until $(\exists u : (u, y) \in D$ *or* $R = \emptyset),$
\\ Postcondition:
$\{G(x) = y\}.$
(The parameter k can take any value in the specified range and, in particular, it can be determined by supercompilation [20, 21].)

Proposition 3. *Inverse Dynamic Programming template is partially correct.*

Proof. One can proceed according to the Floyd-Hoare method [9, 4, 17] and use the following (one and the same) invariant in both loops (i.e. for the external *repeat*-loop and for the internal *exercise*-loop): D is a subset of the graph of G. ■

Proposition 4. *Assume that for some input data the precondition of the inverse Dynamic Programming template is valid and that the input value y belongs to $G(X)$. Then the algorithm generated from the template eventually terminates.*

Proof. A standard way to prove algorithm (and program) termination is via a *potential* (or bound) function [9, 4, 17], i.e. a function that maps states of the algorithm into natural numbers so that every legal loop execution reduces the value of the function. In particular, let $n \in \mathbb{N}$ be an integer such that $y = G(cnt(n))$, let $m = \sum_{0 \le i \le n} |SPP(cnt(i))|$ and let $\pi(D) = m - |D|$ be a potential function; then every legal iteration of any loop of our algorithm reduces the value of this function at least by one. ■

As follows from Propositions 3 and 4, the inverse Dynamic Programming really computes an inverse function for a function defined by the recursive scheme for descending Dynamic Programming.

Let us give an example. It does not make sense to invert function G that solves the Dropping Bricks Puzzle, since this function is not injective[8]. So let us consider a simpler injective function $F : \mathbb{N} \to \mathbb{N}$

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

that computes Fibonacci numbers. Let us assume that *cnt* is the identical function (i.e. enumeration just in the standard order). Then our Inverse Dynamic Programming algorithm gets the following form:

$var\ x, z :\ \mathbb{N};$
$var\ R := \mathbb{N},\ T :\ 2^{\mathbb{N}};$
$var\ D := \emptyset :\ 2^{\mathbb{N}\times\mathbb{N}};$
$var\ k :\ integer;$
$repeat\ z := Fir(R);\ T := [0..z];\ R := R \setminus [0..z];$
$\quad D := D \cup \{(0,1)|\ 0 \in [0..z]\} \cup \{(1,1)\ |\ 1 \in [0..z]\};$
$\quad exercise\ k \in [1..z]\ times :$
$\qquad D :=\ D \cup \{(u,w) \notin D|\ \exists w_1, w_2 :\ (u-1, w_1), (u-2, w_2) \in D,$
$\qquad\qquad (u-1), (u-2) \in [0..z],\ \&\ w = w_1 + w_2\}$
$until\ \exists u : (u, y) \in D;$
$if\ \exists u : (u, y) \in D\ then\ x := (u\ such\ that\ (u, y) \in D)\ else\ loop.$

After some simplification one can get the following algorithm:

$var\ x, z :\ \mathbb{N};$
$var\ T :\ 2^{\mathbb{N}};$
$var\ D := \emptyset :\ 2^{\mathbb{N}\times\mathbb{N}};$
$var\ k :\ integer;$
$z := 0;\ D := \{(0,1), (1,1)\};$

[8] It means that an inverted function will compute a height h for which n droppings *suffice*, but not the *largest* admissible value of h.

$repeat\ z := z + 1;$
$\quad D := D \cup \{(z, w_1 + w_2)\ |\exists w_1, w_2 :\ (z-1, w_1), (z-2, w_2) \in D\};$
$until\ \exists u : (u, y) \in D;$
$if\ \exists u : (u, y) \in D\ then\ x := (u\ such\ that\ (u, y) \in D)\ else\ loop$
that just computes and saves the Fibonacci sequence in the "array" D and checks whether there is y in the array already.

6. Concluding remarks

The author has no intension to make everyone think about Dynamic Programming in terms of fix-point computations. He just believes that the iterative ascending Dynamic Programming template presented in the paper will help to teach and (maybe) automatize Algorithm Design. This approach to teaching Dynamic Programming as well as the approach to backtracking and branch and bound (presented in citeShilov11) have been in use in the Master Program at the Information Technology Department of Novosibirsk State University since 2003.

At the same time, a number of questions concerning further research arise from our study. The first one is related to the inversion of *partial* functions that are defined by the scheme of recursive Dynamic Programming over countable domains. The second one is about the inversion of total functions defined by the scheme over non-countable domains (real number for instant).

Another problem for further research refers to computer architecture for parallel Dynamic Programming. The idea to use *dataflow* architecture is on the surface. A dataflow computer is a collection of *nodes* of different kinds or types that pass *tokens* to each other. Every token comprises *data* and an *address* that define the *type* of the addressee and the *context* (identifying a particular instance of the node of this type).

Let function $G : X \to Y$ be defined by the recursive scheme of descending Dynamic Programming 1, $x \in X$ be a particular argument value, and bas, $spp : X \to 2^X$ be the base and support for G. Then the dataflow graph for computing $G(x)$ consists of nodes of two types, *step* and *start*, that correspond to the elements of $\{v \in spp(x) :\ \neg p(v)\}$ and to the elements of $\{v \in spp(x) :\ p(v)\}$, respectively, and edges (u, v) such that $u \in bas(v)$:

node step $(input :\ token_1, \ldots token_n)$ $\quad result :=\ g(self,\ h_1(token_1.data),$ $\qquad \ldots h_n(token_n.data));$ $\quad if\ self = x$ $\quad then\ send(result)\ to\ host$ $\quad else\ for\ each\ u\ that\ self \in bas(u)$ $\qquad do\ send(result)$ $\qquad\qquad to\ step(context :\ u)$ *end step*	*node start* $\quad result :=\ f(self);$ $\quad if\ self = x$ $\quad then\ send(result)\ to\ host$ $\quad else\ for\ each\ u\ that\ self \in bas(u)$ $\qquad do\ send(result)$ $\qquad\qquad to\ step(context :\ u)$ *end start*

where $self$ represents the context of the node itself and absence of input means that the node is ready for immediate execution. Unfortunately, dataflow parallelism is still *in Virto*, not *in Vita* yet.

Acknowledgments:

The author would like to thank his colleagues Eugene Bodin for the Dropping Bricks Puzzle formulation, Svetlana Shilova for the analytical solution of the puzzle, Vladislav Kuzkokov for the discussions of the puzzle generalization, the anonymous reviewer for useful suggestions on the format and style of the paper and for detecting many conceptual and technical bugs in the draft, and Irina Adrianova for the editorial help and guidance.

References

[1] Aho A.V., Ullman J.D. The Theory of Parsing, Translation, and Compiling (Vol. 1). – Prentice Hall International, 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., Gradel E. (Eds.) Lectures in Game Theory for Computer Scientists. – Cambridge University Press. 2011.

[4] Apt K.R., de Boer F.S., Olderog E.R. Verifcation of Sequential and Concurrent Programs (3rd edition). – Springer, 2009.

[5] Astapov D. Recursion + Memoization = Dynamic Programming // Practice of Functional Programming. – 2009. – N 3. – P. 17–33. – `http://fprog.ru/2009/issue3/` (In Russian).

[6] Bellman R. The theory of dynamic programming // Bulletin of the American Mathematical Society. – 1954. – Vol. 60. – P. 503–516.

[7] Cormen T.H., Leiserson C.E., Rivest R.L., Stein C. Introduction to Algorithms (3-rd edition). – The MIT Press, 2009.

[8] Gimbert H. Games for Verification and Synthesis. Slides for 10th School for young researchers about Modelling and Verifying Parallel processes (MOVEP). — `http://movep.lif.univ-mrs.fr/documents/marta-slides1.pdf`.

[9] Gries D. The Science of Programming. – Springer, 1987.

[10] Greibach S.A. Theory of Program Structures: Schemes, Semantics, Verification. – Springer, 1975. – Lect. Notes Comput. Sci. – Vol. 36.

[11] Kotov V.E., Sabelfeld V.K. Theory of Program Schemata. (Teoria Skhem Programm). – Nauka, 1991 (In Russian).

[12] Knaster B., Tarski A. Un theoreme sur les fonctions d'ensembles // Ann. Soc. Polon. Math. — 1928. — N 6. – P. 133–134.

[13] Lange M., LeißH. To CNF or not to CNF? An Efficient Yet Presentable Version of the CYK Algorithm // Informatica Didactica. — 2009. — N 8. – http://www.informatica-didactica.de/cmsmadesimple/index.php?page=LangeLeiss2009_en.

[14] Paterson M.S., Hewitt C.T. Comperative Schematology // Proc. of the ACM Conf. on Concurrent Systems and Parallel Computation, 1970. – P. 119–127.

[15] Shilov N.V. A note on three Programming Paradigms // 2nd Internat. Valentin Turchin Memorial Workshop on Metacomputation in Russia, 2010. – Ailamazyan Program Systems Institute, Pereslavl-Zalessky, Russia. – P. 173–184.

[16] Shilov N.V. Algorithm Design Template base on Temporal ADT // Proc. of 18th Internat. Symposium on Temporal Representation and Reasoning, 2011. – IEEE Computer Society. – P. 157–162.

[17] Shilov N.V. Introduction to Program Syntax, Semantics and Verification. Novosibirsk State University, 2011. (In Russian). – Draft is available at http://persons.iis.nsk.su/files/persons/pages/langproc.pdf).

[18] Shilov N.V. Inverting Dynamic Programming // Proceedings of the Third International Valentin Turchin Workshop on Metacomputation, 2012. – Publishing House University of Pereslavl. – P. 216–227.

[19] Shilov N.V., Shilova S.O. Bricks and Dynamic Programming // Potential. – 2012. – N 9. – P. 39–44 (In Russian).

[20] Turchin V.F. The concept of a supercompiler // ACM Transactions on Programming Languages and Systems. — 1986. – Vol.8. – N 3. – P. 292–325.

[21] 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 (Sistemnaya Informatika). – Nauka, 1998. – Vol. 6. –. P. 64–89 (In Russian).