Etude on Recursion Elimination by Program Manipulation and Problem Analysis

Nikolay V. Shilov
Innopolis University
Russia
n.shilov@innopolis.ru

Transformation-based program verification was a very important research topic in early years of theory of programming. Great computer scientists contributed to these studies: John McCarthy, Amir Pnueli, Donald Knuth ... Many fascinating examples were examined and resulted in recursion elimination techniques known as *tail-recursion* and as *co-recursion*. In the present paper we examine another (new we believe) example of recursion elimination via program manipulations and problem analysis. The recursion pattern of the study example matches neither tail-recursion nor co-recursion pattern.

1 Introduction

1.1 McCarthy 91 function

Let us start with a short story [7] about the *McCarthy 91 function* $M : \mathbb{N} \to \mathbb{N}$, defined by John McCarthy[1] as a test case for formal verification. The function is defined recursively as

$$M(n) = \begin{cases} n-10, \text{ if } n > 100; \\ M(M(n+11)), \text{ if } n \leq 100. \end{cases}$$

The result of evaluating the function are given by

$$M(n) = \begin{cases} n-10, \text{ if } n > 101; \\ 91, \text{ if } n \leq 101. \end{cases} \tag{1}$$

The function was introduced in papers published by Zohar Manna, Amir Pnueli and John McCarthy in 1970 [15, 16]. These papers represented early developments towards the application of formal methods to program verification. The function has a "complex" recursion pattern (contrasted with simple patterns, such as *recurrence*, *tail-recursion* or *co-recursion*).

Nevertheless the McCarthy 91 function can be computed by an iterative algorithm/program. Really, let us consider an auxiliary recursive function $M_{aux} : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$

$$M_{aux}(n,m) = \begin{cases} n, \text{ if } m = 0; \\ M_{aux}(n-10,\ m-1), \text{ if } n > 100 \text{ and } m > 0; \\ M_{aux}(n+11,\ m+1), \text{ if } n < 100 \text{ and } m > 0. \end{cases}$$

[1] Maybe *the only* Turing Laureate that was employed in Soviet Academy of Sciences, namely — in *Novosibirsk Computing Center* (http://ershov-arc.iis.nsk.su/archive/eaindex.asp?lang=1&did=20184&_ga=1.44945571.687493938.1476117474, accessed April 14, 2014).

Submitted to:
VPT 2017

Then $M(n) = M_{aux}(n,1)$ because of $M_{aux}(n,m) = M^m(n)$ for all $m,n \in \mathbb{N}$ (assuming that $M^0 = (\lambda n \in \mathbb{N}.n)$). Since definition of M_{aux} matches tail-recursion pattern then the McCarthy 91 function can be computed by an iterative algorithm/program (and even by a very efficient *iteration-free* algorithm given by the formula (1)). A formal derivation of an iterative version from the recursive one was given in [20] in 1980 based on the use of continuations.

As the field of Formal Methods advanced, this example appeared repetitively in the research literature. In particular, it was viewed as a "challenge problem" for automated program verification. Donald Knuth generalized the function to include additional parameters [11], formal proofs (using ACL2 theorem prover) that Knuth's generalized function is total can be found in [4, 5].

1.2 Dropping Bricks from a High Tower

We started with a short story about the McCarthy function because we would like to justify our interest to study of the following *Dropping Bricks Problem* (DBP) [18, 19]. (A variant of the problem can be found in [6]).

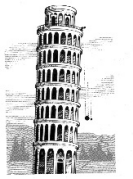

Figure 1: DBP

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. 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. (One may assume that mechanical stability does not change after a safe fall.) How many times do you need to drop bricks, if you have 2 bricks in the stock? What is the optimal number (of droppings) in this case?

Basically, the question 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. Our purpose is to prove that the problem is solved by the following simple formula

$$G(H) = \arg\min n : \ \frac{n \times (n+1)}{2} \geq H \tag{2}$$

that can be implemented as a trivial non-recursive function (i.e. with iterative body) $G_{iter}(H : \mathbb{N})$:

1. *var* $n : \mathbb{N}$;
2. $n := 0$;
3. *while* $\frac{n\times(n+1)}{2} < H$ *do* $n := n+1$;
4. $G_{iter} := n$.

With a purpose to get the above formula (2), let us start with a recursive solution for DBP. This problem is an example of optimization problems. Any optimal method to define 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 crashes after the first dropping (and we have to drop the remaining second brick from the levels 1, 2, $\ldots (h-1)$ in a row),

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 $[(h+1)\ldots H]$),
4. 'max' corresponds to the worst in two cases 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 $G_0 = 0$.

Remark that the 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 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 $G:\mathbb{N}\to\mathbb{N}$ that maps every natural x to G_H and satisfies the following *functional equation* for the *objective function* G:

$$G(x) \;=\; if\ x=0\ then\ 0\ else\ 1+\min_{1\le h\le x}\max\{(h-1),G(x-h)\}. \tag{3}$$

This equation has a unique solution (it follows from the uniqueness of the sequence G_0, G_1, ... G_H, ...). Thus we prove the following proposition.

Proposition 1 *Functional equation (3) has unique solution in* $\mathbb{N}^{\mathbb{N}}$.

Since the left-hand side of this equation has the most general pattern $G(x)$, the equation can be adopted as a recursive definition of a function, i.e. a recursive algorithm presented in a functional pseudo-code.

1.3 DBP from Dynamic Programming Perspective

Dynamic Programming was introduced by Richard Bellman in the 1950s [1] to tackle optimal planning problems. In 1950s the 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*). *Bellman equation* is a functional equality for the objective function that expresses the optimal solution at the "current" state in terms of the optimal solution at next (changed) states. It formalizes a so-called *Bellman Principle of Optimality*: an optimal program (or plan) remains optimal at every stage.

After analysis of Bellman equations for particular problems [3] several versions of a *(recursive template for/of) (descending) dynamic programming* were suggested and examined [18, 19], in the present paper we use the most recent and general one [19]:

$$G(x) \;=\; if\ p(x)\ then\ f(x)\ else\ g\Big(x,\ \Big\{h_i\big(x,G(t_i(x))\big),\ i\in[1..n(x)]\Big\}\Big). \tag{4}$$

We consider the template as a *recursive program scheme* [9, 12, 17], i.e. a recursive control flow structure with *uninterpreted symbols*:

- G is the *main* function symbol representing (after interpretation of *ground* function and predicate symbols) the objective function $G:X\to Y$ for appropriate X and Y;
- p is a ground predicate symbol representing (after interpretation) some *known*[2] predicate $p\subseteq X$;

[2] i.e. that we know how to compute

- f is a ground function symbol representing (after interpretation) some known[2] function $f : X \to Y$;
- g is a ground function symbol representing (after interpretation) some known[2] function $g : X \times Z^* \to X$ for some appropriate Z (with a variable arity $n(x) : X \to \mathbb{N}$);
- all h_i and t_i ($i \in [1..n(x)]$) are ground function symbols representing (after interpretation) some known[2] function $h_i : X \times Y \to Z$, $t_i : X \to X$ ($i \in [1..n(x)]$).

In the sequel we do not make an explicit distinction in notation for symbols and interpreted symbols but just verbal distinction by saying, for example, *symbol g* and *function g*.

Equation (3) for Dropping Bricks Problem is a particular example of functional equation that matches the recursive template for descending dynamic programming (4). In this case we have:

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

A natural question arises: is there a *standard scheme* [9, 12, 17] (i.e. a flowchart with uninterpreted predicate and functional symbols instead of predicate and functions) that is *functionally equivalent* to the recursive scheme (4)? Unfortunately, in general case the answer is negative according to the following proposition proved by M.S. Paterson and C.T. Hewitt [12, 17].

Proposition 2 *The following special case of the recursive template of descending dynamic programming*

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

is not equivalent to any standard program scheme (with fix-size static memory).

This proposition does not mean that dynamic memory (i.e. stack or heap) is *always* required; it just says that for *some* interpretations of *uninterpreted* 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 F can be computed by an iterative program without dynamic memory. For example, Fibonacci numbers

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

matches the pattern in the proposition 2, but just three integer variables suffice to compute Fibonacci numbers by an iterative program.

Thus proposition 2 rules out an opportunity to construct an iterative solution for DBP by specialization [8, 10] of a standard program scheme, but does not prohibit existence of an iterative algorithm for DBP that uses interpreted functions and predicates.

2 Iterative Algorithm for DBP

2.1 Informal Derivation of Iterative Algorithm

Let us start with a look at fig. 2 that depicts an initial part of the graph of G computed according to (3). One can observe that G

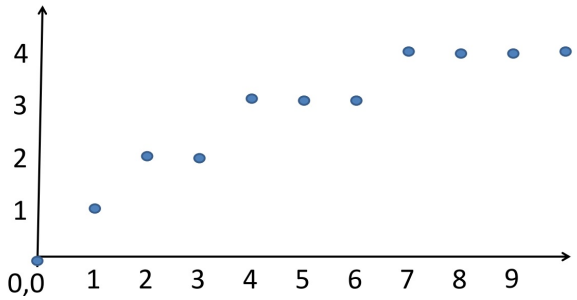

Figure 2: First values of G

- is a non-decreasing function
- has no jumps greater than 1.

Basically these properties follow from semantics of the function G as a solution for DBP but *we do not know how to prove them formally from the equation (3).*

Then let us exercise recursive algorithm (3) manually and symbolically:

$$G(x) = 1 + \min_{1\le h\le x}\max\{(h-1),\, G(x-h)\} =$$
$$= 1 + \min\Big\{\max\{0,\, G(x-1)\},\, \max\{1,\, G(x-2)\},$$
$$\ldots\ldots\ldots\ldots\ldots\ldots$$
$$\max\{(y-2),\, G(x-y-1)\},$$
$$\max\{(\mathbf{y}-\mathbf{1}),\, \mathbf{G(x-y)}\},$$
$$\max\{(y,\, G(x-y+1)\},$$
$$\ldots\ldots\ldots\ldots\ldots\ldots$$
$$\max\{(x-2),\, G(1)\},\, \max\{(x-1),\, G(0)\}\Big\}$$

where the line with **bold** font corresponds to the last value $h \in [1..x]$ such that $(h-1) \le G(x-h)$. Due to the monotonicity property we have for this $\mathbf{y}$

$$G(x) = 1 + G(x-\mathbf{y}). \tag{5}$$

Due to monotonicity and jump properties we have

$$G(x-\mathbf{y}) = (\mathbf{y}-1) \text{ or } G(x-\mathbf{y}) = \mathbf{y};$$

let us accept the former option and rule out the late (*but we can not prove why we may do it*):

$$G(x-\mathbf{y}) = (\mathbf{y}-1). \tag{6}$$

Now, for the technical convenience, let a be $(x-\mathbf{y})$, b — $(\mathbf{y}-1)$; then $x = (a+b+1)$ and (according to (5) and (6))

$$G(a) = b \implies G(a+b+1) = (b+1). \tag{7}$$

Together with equality $G(0) = 0$ it leads (by induction) to the following equality

$$G(\sum_{h=1}^{h=n} h) = n. \tag{8}$$

2.2 Proving Optimality

Formula (8) leads to the following procedure $Strength(Hight : \mathbb{N})$ to define mechanical strength of bricks using 2 identical bricks (referred as *the first* and *the second*):

1. $var\ n,\ step,\ Current,\ Next : \mathbb{N}$;
2. let $n := \arg\min n : \frac{n\times(n+1)}{2} \geq H$;
3. let $step := n$ and $Current := 0$;
4. dropping the first brick:
   while $Current < Hight$ and $step > 0$ do
   (a) let $Next := \min\{Hight,\ (Current + step)\}$
   and drop the first brick from the *Next* level;
   (b) if the drop was unsafe (i.e. the first brick crashes)
   then break the loop and go to 5 (dropping the second brick);
   (c) let $Current := Next$ and $step := (step - 1)$;
5. dropping the second brick:
   let $Current := (Current + 1)$;
6. while $Current < Next$ do
   (a) drop the second brick from the *Current* level;
   (b) if the drop was unsafe (i.e. the second brick crashes)
   then break the loop and go to 7 (report the strength);
   (c) let $Current := (Current + 1)$;
7. report the strength: $(Current - 1)$.

To explain the idea of the procedure $Strength(H)$ (where $H \in \mathbb{N}$), let us assume that the height the tower H is exactly the sum of a decrising arithmetic progression n, $(n-1)$, ... 2, 1. Then the procedure divides the tower onto n layers of heights $step_1 = n$, $step_2 = (n-1)$, ... $step_{(n-1)} = 2$ and $step_n = 1$. (In the left part of fig.3 one can see a tower of hight 10 divided on 4 layers of heights 4, 3, 2 and 1.)

The first loop in the procedure body drops the first brick in a sequence (while it is safe) from the (top of) layers at levels n, $\big(n+(n-1)\big)$, $\Big(\big(n+(n-1)\big)+(n-2)\Big)$, ... until it breaks after dropping from the top of some layer $k \geq 1$ from the level $\Big(\ldots\Big(\big(n+(n-1)\big)+(n-2)\Big)\ldots+(n-(k-1))\Big)$. (In particular, $k = 2$ in the right part of fig.3.)

The second loop in the procedure body examines (using the second brick) one by one (while the brick is safe) all levels

$$\text{from } 1+\Big(\ldots\Big(\big(n+(n-1)\big)+(n-2)\Big)\ldots+(n-(k-2))\Big)$$

$$\text{to } \Big(\ldots\Big(\big(n+(n-1)\big)+(n-2)\Big)\ldots+(n-(k-1))\Big)-1$$

of the layer $k \geq 1$ (from the top of which the first brick fell down and broke). (Two levels 5 and 6 in the example in the right part of fig.3.)

The mechanical strength of the bricks is the last level from which the second brick was safely dropped. (Level 5 in the example in fig.3.)

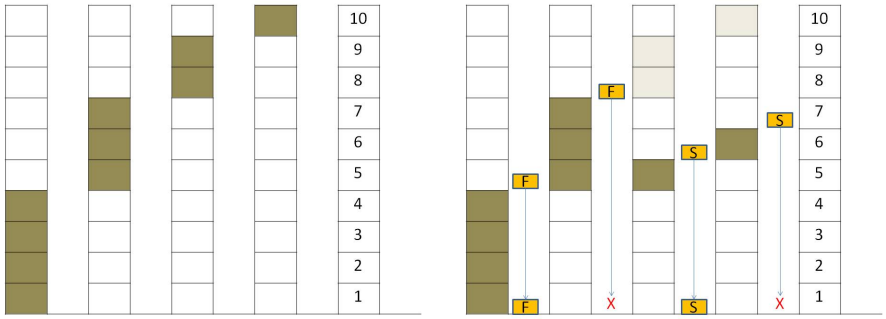

Figure 3: The layers (left) and an exercise (right) of the procedure *Strength* for the tower of height 10 and two bricks (F and S) of mechanical strength 5

Optimality (i.e. minimality of droppings) of the procedure *Strength* can be proved by contradiction. Really, let us assume in contrary that for some $H \in \mathbb{N}$ another method gives better number m of droppings in the worst case. Better number this time means that $m < n = \arg\min n : \frac{n\times(n+1)}{2} \geq H$. This method also divide the tower on layers from top of which the first brick have to be dropped (according to the method): assuming $Level_0 = 0$,

- the top of the first layer is $Level_1 = Level_0 + m_1$,
- the top of the second layer is $Level_2 = (Level_1 + m_2)$,
- .....
- the top of the last layer is $Level_{last} = Level_{last-1} + m_{last} = H$.

Then we have:

- $last \leq m$, because the first brick can survive all m drops;
- $m_1 \leq m$, because the first brick can break after the *first* drop;
- $m_2 \leq (m-1)$, because the first brick can break after the *second* drop;
- .....
- $m_k \leq (m-(k-1))$, because the first brick can break after *k-th* drop, $(1 \leq k \leq last)$;
- .....
- $m_{last} \leq (m-(last-1))$, because the first brick can break after the *last* its drop.

Hence we have:

$$H = m_1 + m_2 + \ldots m_{last} \leq m + (m-1) + \ldots (m-(last-1)) \leq \sum_{k=1}^{k=m} k \leq \sum_{k=1}^{k=n} k;$$

at the same time (according to choice of n)

$$n = \arg\min n : \frac{n\times(n+1)}{2} \geq H;$$

it implies that $m = n$. — Contradiction with the assumption $m < n$. Thus we prove the following proposition.

Proposition 3 *Procedure Stregth implements an optimal (in sense of number of droppings) method to define mechanical strength of bricks using 2 bricks: for any given $H \in \mathbb{N}$ it defines mechanical strength dropping bricks*

$$\arg\min n : \ \frac{n \times (n+1)}{2} \geq H$$

times at most (and this upper bound is exact).

According to proposition 1, functional equation 3 has unique solution in $\mathbb{N}^{\mathbb{N}}$ that computes the optimal number of droppings that is sufficient to define mechanical strength; according to proposition 3, this number is given by formula 2. Thus we prove the following proposition.

Proposition 4 *Functional equation (3)*

$$G(x) \ = \ if \ x = 0 \ then \ 0 \ else \ 1 + \min_{1 \leq h \leq x} \max\{(h-1), G(x-h)\}$$

has unique solution in $\mathbb{N}^{\mathbb{N}}$ given by

$$\lambda x \in \mathbb{N}. \ \arg\min n : \ \frac{n \times (n+1)}{2} \geq x.$$

3 On Iterative Dynamic Programming

3.1 Template with Associative Array

Let us consider a function $G : X \to Y$ that is defined by an interpreted recursive scheme (4) of Dynamic Programming. Let us define two sets $bas(v), spp(v) \subseteq X$:

- base $bas(v) = \ if \ p(v) \ then \ \emptyset \ else \ \{t_i(v) : i \in [1..n(v)]\} \subseteq X$ comprises all values that are immediatly needed to compute $G(v)$;
- support $spp(v)$ be the set of all values that appear in the call-tree of $G(v)$.

Remark that $bas(v)$ is always finite and support $spp(v)$ is finite iff G is defined on v; the support may be computed by the following recursive algorithm:

$$spp(x) \ = \ if \ p(x) \ then \ \{x\} \ else \ \{x\} \cup (\bigcup_{y \in bas(x)} spp(y)). \tag{9}$$

Some variants of template for iterative ascending dynamic programming were already suggested, specified and verified in [19]. Let us suggest, specify and verify a new, more concise variant that we call in this paper *iterative template for/of (ascending) dynamic programming*:

- Template Applicability Conditions *TAC*:
  1. I is an interpretation for ground symbols in scheme (4);
  2. $n : X \to \mathbf{N}$ is arity function of interpreted g;
  3. $G : X \to Y$ is the objective function, i.e. a solution of interpreted by I scheme (4);
  4. $t_1, \ldots : X \to X$ are functions that computes the base;
  5. $spp : X \to 2^X$ is the support function for G;
  6. $NiX \notin X$ is the distinguishable indefinite value[3] for X;

[3] *NiX — Non in X*, similarly to *Non a Number* — `NaN`.

- Template Pseudo-Code TPC:
  1. $VAR\ LUT\ : Y\ array\ of\ spp(v)$;
  2. $LUT := \ array\ filled\ by\ NiX$;
  3. $for\ all\ u \in spp(v)\ do\ if\ p(u)\ then\ LUT[u] := f(u)$;
  4. $while\ LUT[v] = NiX\ do$
     $let\ u \in spp(v)\ be\ any\ element\ in\ spp(v)$
     $\quad such\ that\ LUT[u] = NiX\ and$
     $\quad LUT[t_i(u)] \neq NiX\ for\ all\ i \in [1..n(u)]$
     $in\ LUT[u] := \ g\left(u,\ \left\{h_i\big(u, LUT[t_i(u)]\big),\ i \in [1..n(u)]\right\}\right)$.

Remark, that the template is not a *standard program scheme*, but scheme augmented by *associative array* LUT of values in Y indexed by values in $spp(v)$.

Proposition 5 *Assuming Template Applicability Conditions TAC the following holds for every $v \in X$:*

1. *if $G(v)$ is defined then interpreted template TPC terminates after $|spp(v)|$ iterations of both loops, and $LUT[v] = G(v)$ by termination;*
2. *if $G(v)$ is not defined then interpreted template TPC never terminates.*

The proof for all propositions 5 presented here are very easy and almost straightforward. The advantage of TPC is the use of just one-time allocated *associative array* instead of a *stack* required to translate general recursion.

3.2 Concluding Remarks

Use of arrays for efficient translation of recursive functions of integer argument was suggested first (up to our knowledge) in [2]. In the cited paper this technique of recursion implementation is called *production mechanism*. The essence of the production mechanism consists in support evaluation (that is a set of integers), array declaration with the index range that subsumes the support, and fill-in this array in bottom-up (i.e. ascending) manner by values of the recursive function.

Use of auxiliary array has been studied also in [14], and more broadly in [13]. These previous works do not use template (like the recursive template (4) for dynamic programming) but can save space asymptotically over the proposed template. (For example, computing length of longest common subsequence takes only linear space using the previous method, instead of quadratic space using the template.)

Nevertheless a novelty of our study consists in use of templates (understood as semi-interpreted program schemata) and sematic properties that constraint interpretations where recursive programs matching the template may be computed efficiently by iterative imperative programs (with an associative array).

References

[1] R. Bellman (1954): *The theory of dynamic programming*. Bulletin of the American Mathematical Society 60, pp. 503–516.

[2] G. Berry (1976): *Bottom-up computation of recursive programs*. RAIRO — Informatique Théorique et Applications (Theoretical Informatics and Applications 10(3), pp. 47–82.

[3] T.H. Cormen, C.E. Leiserson, R.L. Rivest & C. Stein (2009): *Introduction to Algorithms*, third edition. The MIT Press.

[4] J. Cowles (2000): *Knuth's generalization of McCarthy's 91 function*, pp. 283–299. Kluwer Academic Publishers.

[5] J. Cowles & R. Gamboa (2004): *Contributions to the Theory of Tail Recursive Functions*. Available at `http://www.cs.uwyo.edu/~ruben/static/pdf/tailrec.pdf`. Accessed April 14, 2017.

[6] From Wikipedia the free encyclopedia: *Dynamic Programming*. Available at `http://en.wikipedia.org/wiki/Dynamic_programming#Egg_dropping_puzzl`. Accessed April 14, 2017.

[7] From Wikipedia the free encyclopedia: *McCarthy 91 function*. Available at `https://en.wikipedia.org/wiki/McCarthy_91_function`. Accessed April 14, 2017.

[8] A.P. Ershov (1982): *Mixed computation: potential applications and problems for study*. *Theor. Comp. Sci.* 18(1), pp. 41–67.

[9] S.A. Greibach (1975): *Theory of Program Structures: Schemes, Semantics, Verification*. *Lecture Notes in Computer Science* 36, Springer.

[10] N.D. Jones, C.K. Gomard & P. Sestoft (1993): *Partial Evaluation and Automatic Program Generation.)*. Prentice Hall International.

[11] D.E. Knuth (1991): *Textbook Examples of Recursion*. Available at `arXiv:cs/9301113`. Accessed April 14, 2017.

[12] V.E. Kotov & V.K. Sabelfeld (1991): *Theory of Program Schemata. (Teoria Skhem Programm.)*. Science (Nauka), Moscow. (In Russian).

[13] Y.A. Liu (2013): *Systematic Program Design: From Clarity to Efficiency*. Cambridge University Press.

[14] Y.A. Liu & S.D. Stoller (2002): *Program optimization using indexed and recursive data structures*. In: *Proceedings of the 2002 ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation*, ACM, pp. 108–118.

[15] Z. Manna & J. McCarthy (1970): *Properties of programs and partial function logic*. *Machine Intelligence* 5, pp. 79–98.

[16] Z. Manna & A. Pnueli (1970): *Formalization of Properties of Functional Programs*. *Journal of the ACM* 17(3), pp. 555–569.

[17] M.S. Paterson & C.T. Hewitt (1970): *Comperative Schematology*. In: *Proc. of the ACM Conf. on Concurrent Systems and Parallel Computation*, ACM, pp. 119–127.

[18] N.V. Shilov (2012): *Unifying Dynamic Programming Design Patterns*. *Bulletin of the Novosibirsk Computing Center (Series: Computer Science, IIS Special Issue)* 34, pp. 135–156.

[19] N.V. Shilov (2016): *Algorithm Design Patterns: Program Theory Perspective*. In: *Proc. of Fifth Int. Valentin Turchin Workshop on Metacomputation (META-2016)*, University of Pereslavl, pp. 170–181.

[20] M. Wand (1980): *Continuation-Based Program Transformation Strategies*. *Journal of the ACM* 27(1), pp. 164–180.