Моделирование и анализ информационных систем. Т. 25, № 5 (2018), с. 549–560
Modeling and Analysis of Information Systems. Vol. 25, No 5 (2018), pp. 549–560

DOI: 10.18255/1818-1015-2018-5-549-560

UDC 519.68

Etude on Recursion Elimination

Shilov N. V.

Received September 26, 2018

Abstract. Transformation-based program verification was a very important 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 *co-recursion*. In the paper, we examine just a single example (but new we hope) of recursion elimination via program manipulations and problem analysis. The recursion pattern of the example matches descending dynamic programming but is neither tail-recursion nor co-recursion pattern. Also, the example may be considered from different perspectives: as a transformation of a descending dynamic programming to ascending one (with a fixed-size static memory), or as a proof of the functional equivalence between recursive and iterative programs (that can later serve as a case-study for automatic theorem proving), or just as a fascinating algorithmic puzzle for fun and exercising in algorithm design, analysis, and verification. The article is published in the author's wording.

Keywords: recursive and standard program schemata, recursive and iterative programs, functional equivalence of programs and program schemata, ascending and descending dynamic programming, recursion elimination, static and dynamic memory, associative and standard arrays

For citation: Shilov N. V., "Etude on Recursion Elimination", *Modeling and Analysis of Information Systems*, **25**:5 (2018), 549–560.

On the authors:
Nikolay V. Shilov, orcid.org/0000-0001-7515-9647, PhD,
Autonomous noncommercial organization of higher education "Innopolis University"
1 Universitetskaya str., Innopolis, Tatarstan Republic, 420500, Russia, e-mail: shiloviis@mail.ru

1. Introduction

1.1. McCarthy 91 function

We would like to start with a short story about the *McCarthy 91 function* that follows (in principle) the corresponding article [21] "*From Wikipedia, the free encyclopedia*".

The function $M : \mathbb{N} \to \mathbb{N}$ is a recursive function, defined by John McCarthy[1] as a test case for formal verification within computer science. The function is defined as

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

[1]Maybe *the only* Turing Laureate that was employed in Soviet Academy of Sciences, namely *Novosibirsk Computing Center* (`http://ershov-arc.iis.nsk.su/archive/eaindex.asp?lang=1&did=20184&_ga=1.44945571.687493938.1476117474`, accessed September 26, 2018).

The results 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 [16, 15]. 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 $M_{aux} : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$

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

Then $M(n) = M_{aux}(n, 1)$ because of $M_{aux}(n, m) = M^m(n) = \underbrace{M(\ldots M}_{m-times}(n)\ldots)$ 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 (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 is 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. Hull Strength Puzzle

We started with a short story about the McCarthy function because we would like to justify our interest to study of translation of other examples functional/recursive programs into iterative algorithms/programs in general and the following problem[2] that we call in the sequel *Hull Strength Puzzle* (HSP).

Let us characterize the mechanical stability (strength) of a hull of a mobile phone by an integer h that is equal to the height (in meters) safe for the case to fall down, while height $(h + 1)$ meters is unsafe (i.e. the brick breaks). You have to determine the stability of hulls 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 hulls, if you have 2 hulls in the stock? What is the optimal number (of droppings) in this case?

[2]The problem formulation is just a literary version of the formulation of the *Dropping Bricks Problem* used in [18, 19], another variant of the problem formulation — *Egg dropping puzzle* — can be found in Wikipedia article on *Dynamic Programming* at https://en.wikipedia.org/wiki/Dynamic_programming#Egg_dropping_puzzle (accessed September 26, 2018).

Basically, the question 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.

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 HSP. 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 phone 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 hull of the first phone breaks after the first dropping (and we have to drop the remaining second phone from the levels 1, 2, $\ldots (h-1)$ in a series),
3. G_{H-h} corresponds to the case when the hull of the first phone is safe after the first dropping (and we have to define stability by dropping the pair of phones 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:

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

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_x 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 as it follows from the uniqueness of the sequence G_0, G_1, ... G_H, ... Let us summarize the above discussion as the following proposition.

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

Moreover we can go further: the equation (3) can be adopted as a recursive definition of a function, i.e. a recursive algorithm presented in a functional pseudo-code.

1.3. A Special Case of Dynamic Programming

Dynamic Programming was introduced by Richard Bellman in the 1950s [2] to tackle optimal planning problems. At this time, 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 recursive functional equality for the objective function that expresses the optimal solution at the "current" state in terms of optimal solutions 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 [6] several versions of a *(recursive template for/of) (descending) dynamic programming* were suggested and examined. 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* functional symbol representing (after interpretation of *base* functional and predicate symbol) the objective function $G : X \to Y$ for some X and Y;
- p is a basic predicate symbol representing (after interpretation) some *known*[3] predicate $p \subseteq X$;
- f is a basic functional symbol representing (after interpretation) some known[3] function $f : X \to Y$;
- g is a basic functional symbol representing (after interpretation) some known[3] function $g : X \times Z^* \to X$ for some appropriate Z (with a variable arity $n(x) : X \to \mathbb{N}$);

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

- all h_i and t_i $(i \in [1..n(x)])$ are basic functional symbols representing (after interpretation) some known[3] function $h_i : X \times Y \to Z$, $t_i : X \to X$ $(i \in [1..n(x)])$.

In the sequel 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 Hull Strength Puzzle is a particular example of functional equation that matches the recursive template for descending dynamic programming (4). In the 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 $\lambda x.\lambda w_1 \dots \lambda w_n.(\min_{1\le i\le x} w_i)$ is interpretation for g.

A natural question arises: maybe there exists 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 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*

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

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

This proposition does not mean that (potentially) unbounded memory (e.g. system stack or dynamic 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 unbounded memory. For example, Fibonacci numbers

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

matches the pattern of scheme in the above proposition 2, but just three integer variables suffice to compute it by an iterative program.

Thus proposition 2 rules out an opportunity to get iterative solution for Hull Strength Puzzle by specialization [8, 10] of a standard program scheme equivalent to the recursive scheme (4). But this proposition does not prohibit existence of an iterative algorithm for HSP that uses interpreted functions and predicates.

2. Iterative Algorithm for HSP

2.1. Toward Iterative Algorithm

Let us present some (not very formal) derivation of the formula (2) for Hull Strength Puzzle and start with a look at Fig. 1 that depicts an initial part of the graph of G computed according to (3). One can observe that

(monotonicity): G is a non-decreasing function,

(jump property): it has no jumps greater 1.

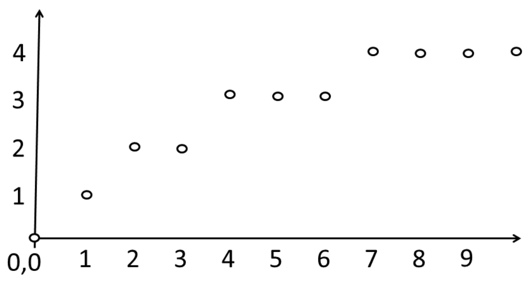

Fig. 1. First values of the function G

Basically these *monotonicity property* and *jump property* follow from semantics of the function G as a solution for HSP, but *we do not know how to prove them formally from the equation (3)*.

Then let us proceed symbolically $G(x)$ according to recursive algorithm (3):

$$\begin{aligned} G(x) \;=\; 1 \;+\; \min\nolimits_{1\le h\le x}\max\{(h-1),\; G(x-h)\} \;=& \\ = 1 \;+\; \min\Big\{\max\{0,\; G(x-1)\},\; \max\{1,\; G(x-2)\},& \\ \dots\dots\dots\dots\dots\dots& \\ \max\{(y-2),\; G(x-y-1)\},& \\ \max\{(\mathbf{y-1}),\; \mathbf{G(x-y)}\},& \\ \max\{(y,\; G(x-y+1)\},& \\ \dots\dots\dots\dots\dots\dots& \\ \max\{(x-2),\; G(1)\},\; \max\{(x-1),\; G(0)\}\Big\}& \end{aligned}$$

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

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

Due to monotonicity and jump properties we have

$$\left[\begin{array}{l} \text{either } G(x-y) = y \\ \text{or } G(x-y) = (y-1) \end{array}\right. ;$$

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

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

Now, for the technical convenience, let a be $(x-y)$, b — $(y-1)$; then $x = (a+b+1)$ and (5) and (6) lead to the equality

$$G(a) = b \text{ inplies } G(a+b+1) \;=\; (b+1). \tag{7}$$

Together with another equality $G(0) = 0$ it leads to the following equality (that can be proved by induction)

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

2.2. An Optimal Procedure for Mechanical Strength

Formula (8) leads also to the following procedure $Strength(Hight : \mathbb{N})$ to define mechanical strength of the hull using 2 identical mobile phones (named the first and the second in the sequel):

```
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 phone:
   while $Current < Hight$ and $step > 0$ do
     (a) let $Next := \min\{Hight,\ (Current + step)\}$
         and drop the first phone from the $Next$ level;
     (b) if the drop was unsafe (i.e. the hull of the first phone breaks)
                then break the loop and go to 5 (dropping the second phone);
     (c) let $Current := Next$ and $step := (step - 1)$;
5. dropping the second phone:
   let $Current := (Current + 1)$;
6. while $Current < Next$ do
     (a) drop the second phone from the $Current$ level;
     (b) if the drop was unsafe (i.e. the hull of the second phone breaks)
                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 of the tower H is exactly the sum of an 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 $step_n = 1$. (For example, in the left part of Fig. 2 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 prescribes to drop the first phone in a sequence (while it is safe) from the (top of) layers at levels n, $(n+(n-1))$, $\Big((n+(n-1))+(n-2)\Big)$, ... until it breaks after dropping from the top of some layer $k \geq 1$ from the level $\Bigg(\dots\Big((n+(n-1))+(n-2)\Big)\dots+(n-(k-1))\Bigg)$. (In the exercise of the procedure in the right part of Fig. 2 $k=2$.)

The second loop in the procedure prescribes to use the second phone moving one by one (while the phone is safe) all levels from $\Bigg(\dots\Big((n+(n-1))+(n-2)\Big)\dots+1)\Bigg)$ to $\Bigg(\dots\Big((n+(n-1))+(n-2)\Big)\dots+(n-(k-2))\Bigg)$ of the layer $k \geq 1$ (from the top of which the first phone fell down and broke). (In the exercise of the procedure in the right part of Fig. 2 two levels — 5 and 6 — were examined.)

The mechanical strength of the hull is the last level from which the second brick was safely dropped. (In the exercise of the procedure in the right part of Fig. 2 it is level 5.)

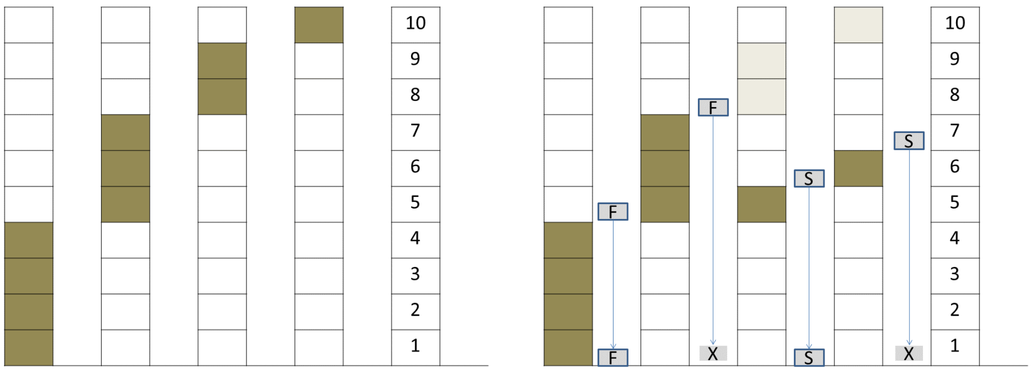

Fig. 2. The layers (left) and an exercise (right) of the procedure *Strength* for the tower of height 10 and two bricks (F and S) of mechanical strength 5

Optimality (i.e. the 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 phone 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 $Level_{last} = Level_{last-1} + m_{last} = H$.

Remark that $last \leq m$, because the first phone can survive all m droppings. Then we have:

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

Hence we have:

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

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 the strength. Due to this uniqueness and according to proposition 3, this solution is defined by equality (2.) and $G = G_{iter}$.

3. Conclusion: Towards Formal Verification

Let us start with a summary of a contribution of this paper.

- The paper discusses a so-called Hull Strength Puzzle (see subsection 1.2.) and how to eliminate recursion and build an iterative algorithm to solve the problem.
- The problem under study is an instance of so-called *learning problem* to determine the function in some family that has certain properties by testing (querying) the function several times.
- The recursive solution of the problem is a particular instance of dynamic programming and matches descending dynamic programming template (see subsection 1.3.).
- Unfortunately, the descending dynamic programming template is not equivalent to any fixed standard program scheme (see subsection 1.3.) and hence iterative solution for the problem can not result from a general one by program specialization [8, 10].
- Also, the recursive solution matches neither tail-recursion nor recurrent pattern that can be converted into iterative algorithms by well-known techniques [11].
- We derived a candidate for iterative solution for Hull Strength Puzzle by some program manipulations (basically, loop unfolding) and (not-very sound) semantic analysis of the unfolded loop (see subsection 2.1.).
- Finally we give (see subsection 2.2.a round-about (and very much) human-oriented proof of correctness of the iterative algorithm for Hull Strength Puzzle (using an optimal method to define mechanical strength of the bricks).

Some topics for further studies are presented below (from the nearest to that which require more time).

- To prove using a proof-assistance (ACL2 most probably) that iterative and recursive definitions for the function G (see subsection 1.2.) are equivalent.
- To investigate how to generalize the pattern of the recursive function and very particular manipulations used/presented in this paper for recursion elimination in more general cases.
- Investigate methods to find recursive patterns admitting recursion elimination. Maybe, machine learning can help to advance in this direction.
- To design and implement a plugin for some IDE (Integrated Development Environment) that analyses program code to find recursive patterns admitting recursion elimination and eliminates these cases of recursion at object code level.

We would like to conclude with some references to related research. Transformation approach to efficient programs is under study for high-level functional programming languages [7]. Use of integer arrays for efficient recursion elimination for functions of integer argument was suggested first (up to our knowledge) in [3] and use of auxiliary (associative) arrays for more general recursion elimination was studied later in [13], and more broadly in [14].

References

[1] Olver P. J., *Applications of Lie groups to differential equations*, Springer-Verlag, New York, 1993.

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

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

[4] Cowles J., *Computer-Aided reasoning: ACL2 case studies*, Kluwer Academic Publishers, 2000.

[5] Cowles J., Gamboa R., "Contributions to the Theory of Tail Recursive Functions", 2004, http://www.cs.uwyo.edu/~ruben/static/pdf/tailrec.pdf, accessed September 26, 2018.

[6] Cormen T. H. et al., *Introduction to Algorithms (3rd ed.)*, The MIT Press, 2009.

[7] De Moor O., Sittampalam G., "Generic Program Transformation", Lecture Notes in Computer Science, **1608**, 1999, 116–149.

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

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

[10] Jones N. D. et al., *Partial Evaluation and Automatic Program Generation*, Prentice Hall International, 1993.

[11] Knuth D. E., *Textbook Examples of Recursion*, 1991, arXiv:cs/9301113[cs.CC], accessed September 26, 2018.

[12] Kotov V. E., Sabelfeld V. K., *Theoriya Schem Programm*, Nauka, 1991.

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

[14] Liu Y. A., *Systematic Program Design: From Clarity to Efficiency*, Cambridge University Press, 2013.

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

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

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

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

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

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

[21] “McCarthy 91 function”, https://en.wikipedia.org/wiki/McCarthy_91_function, accessed September 26, 2018.

Шилов Н. В., "Этюд об устранении рекурсии", *Моделирование и анализ информационных систем*, **25**:5 (2018), 549–560.

DOI: 10.18255/1818-1015-2018-5-549-560

Аннотация. Трансформационный подход к верификации программ был очень популярной темой исследований в первые десятилетия теории программирования. Многие выдающиеся пионеры теории программирования внесли свой вклад в разработку данного направления исследований: Джон Маккарти, Амир Пнуели, Дональд Кнут ... Много интересных примеров трансформационного подхода было тщательно изучено, что привело к методам устранения рекурсии, известным как *хвостовая рекурсия* и как *ко-рекурсия.* В данной работе мы подробно исследуем (мы надеемся, новый) пример устранения рекурсии, основанный на трансформациях программы и анализе задачи, решаемой этой программой. Наш пример является частным случаем нисходящего динамического программирования, но не является ни примером хвостовой рекурсии, ни ко-рекурсии. Этот пример можно рассмотреть с разных точек зрения: как пример преобразования нисходящего динамического программирования к восходящему (с использованием только статической памяти фиксированного размера), или как доказательство функциональной эквивалентности между рекурсивной и итеративной программами (которое в дальнейшем может послужить примером для автоматического доказательства), или как захватывающую алгоритмическую головоломку либо задачу дизайна, анализа и верификации алгоритмов. Статья публикуется в авторской редакции.

Ключевые слова: рекурсивные и стандартные схемы программ, рекурсивные и итеративные программы, функциональная эквивалентность программ и схем программ, восходящее и нисходящее динамическое программирование, устранение рекурсии, ассоциативные и стандартные массивы, статическая и динамическая память

Об авторах:
Шилов Николай Вячеславович, orcid.org/0000-0001-7515-9647, канд. физ.-мат. наук, доцент,
Автономная некоммерческая организация высшего образования “Университет Иннополис”,
ул. Университетская, 1, г. Иннополис, Республика Татарстан, 420500, Россия, e-mail: shiloviis@mail.ru