Study of Recursion Elimination for a Class of Semi-Interpreted Recursive Program Schemata

Nikolay V. Shilov

Innopolis University, Innopolis, Republic of Tatarstan, Russia
`shiloviis@mail.ru`

Abstract

We study templates (i.e. control flow structures with uninterpreted functional and predicate symbols commonly known as program schemata) for *descending* and *ascending* dynamic programming, discuss these templates from programming theory perspective in terms of translation of recursive program schemata to iterative ones with or without dynamic memory, suggested sufficient conditions when the recursive template can be translated into iterative program schemata with fix-size static memory.

More than 50 years passed since the "Golden Age" of Theory of Program Schemata in 1960-70's. Great computer scientists contributed to these studies: John McCarthy, Edsger Dijkstra, Donald Knuth, Amir Pnueli... Studies of `go-to` elimination (structured program Böhm-Jacopini theorem about a translation of spaghetti-like iterative code to more understandable and easier to verify iterative code) and recursion elimination (i.e. how to translate recursive program schemata and programs to iterative ones) were very popular in 1960-1970's [4]. Recursion elimination was very popular because it is about translation from easier to design and verify declarative code to more efficient imperative code. Many fascinating examples of recursion elimination have been examined [3, 2, 7] (e.g. *tail-recursion* that is basically a recursive variant of `go-to`). In the paper we study a recursion pattern that doesn't match the tail-recursion, but matches well the pattern of Bellman equation, a general form for recursive dynamic programming. We study this pattern of recursive dynamic programming as a template (i.e. uninterpreted or semi-interpreted program scheme with a variable arity of symbols/functions/predicates) [6], discuss sufficient conditions for the interpretation of functional and predicate symbols when the recursive scheme may be translated to iterative schemata with (i) an associative array with a pre-computed size, (ii) an integer array with pre-computed size, and (iii) a fix-size static memory.

Dynamic Programming was introduced by Richard Bellman in the 1950s to tackle optimal planning problems. *Bellman equation* is a name for 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.* In the present paper we study a class of Bellman equations that matches the following recursive pattern:

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

We consider the pattern as a *recursive program scheme* (or template) [6], i.e. a recursive control flow structure with *uninterpreted symbols*:

- G is the *main* functional symbol representing (after interpretation of *ground* functional and predicate symbol) the objective function $G : X \to Y$ for some X and Y;

- p is a ground predicate symbol representing (after interpretation) some *known*[1] predicate $p \subseteq X$;
- f is a ground functional symbol representing (after interpretation) some known[1] function $f : X \to Y$;
- g is a ground functional symbol representing (after interpretation) some known[1] 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 functional symbols representing (after interpretation) some known[1] 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.

Let us consider a function $G : X \to Y$ that is defined by the interpreted recursive scheme (1). 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 immediately needed to compute $G(v)$;
- support $spp(v)$ is the set of all values that appear in the call-tree of $G(v)$.

Note that $bas(v)$ is always finite and if G is defined on v then the support $spp(v)$ is finite. When $G(v)$ is defined, the support can be computed by the following algorithm:

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

Let us specify and verify the following *iterative template for/of (ascending) dynamic programming*:

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

```
1. $VAR\ LUT\ \ : assosiative\ array\ indexed\ by\ spp(v)\ with\ values\ in\ Y$;
2. $LUT := \ array\ filled\ by\ NiX$;
3. $for\ all\ u \in spp(v)\ do\ if\ p(u)\ then\ LUT[u] := f(u)$;
```

[1] i.e. that we know how to compute
[2] *NiX* — *Non in X*, similarly to *Non a Number* — `NaN`.

```
4. $while\ LUT[v] = NiX\ do$
   $let\ u \in spp(v)\ be\ any\ element\ in\ spp(v)$
       $such\ that\ LUT[u] = NiX\ and$
       $LUT[t_i(u)] \neq NiX\ for\ all\ i \in [1..n(u)]$
   $in\ LUT[u] := g\Big(u,\ \Big\{h_i\big(u, LUT[t_i(u)]\big),\ i \in [1..n(u)]\Big\}\Big).$
```

Note that the template is not a *standard program scheme*, but a scheme augmented by *associative array* (namely LUT).

Proposition 1. *Assuming 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 advantage of TPC is the use of an associative array that is allocated once instead of a stack, which is required to translate general recursion. Nevertheless, a natural question arises: is a finite static memory sufficient when computing this function? Unfortunately, in general, the answer is no according to the following proposition by M.S. Paterson and C.T. Hewitt [6].

Proposition 2. *The following special case of the recursive template (1)*

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

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

Proposition does not imply that dynamic memory 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 with a finite static 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 the scheme in the above proposition 2, but just three integer variables suffice to compute it by an iterative program.

The following proposition states sufficient conditions when a finite static memory suffices to compute the recursive function (1).

Proposition 3. *Assume that TAC holds altogether with the following additional conditions:*

- *arity function $n : X \to \mathbb{N}$ is some constant $n \in \mathbb{N}$;*
- *base functions $t_1, \ldots\ t_n$ are interpreted in such a way that t_1 is invertible and $t_i = (t_1)^i$ for all $i \in [1..n]$;*
- *interpreted predicate p is t_1-closed in the following sense: $p(u) \Rightarrow p(t_1(u))$ for all $u \in X$.*

Let $m \in \mathbb{N}$ be number of static variables that suffice to implement imperative iterative algorithms to compute interpreted ground predicate and functions p, f, h_i ($i \in [1..n]$), t_1 and t_1^- for any input value. Then the objective function G may be computed by an imperative iterative algorithm with $2n + m + 2$ static variables.

(Let us skip a proof of the statement because of space limitations.)

To the best of our knowledge, use of integer arrays for efficient translation of recursive functions of integer argument was suggested first in [1]. In the cited paper this technique of recursion implementation was called *production mechanism*. The essence of the production mechanism consists in support evaluation (that is a set of integers), array declaration with a proper index range, and fill-in this array in bottom-up (i.e. ascending) manner by values of the objective function. Use of auxiliary array was studied also in [5]. The book [5] doesn't use templates but translation techniques asymptotically but is more space efficient that our approach. (For example, if to use techniques from [5], then the length of the longest common subsequence can be computed in linear space, while our approach needs a quadratic space.)

Nevertheless, a novelty of our study consists in the use of templates (understood as semi-interpreted program schemata) and sematic sufficient conditions that allow recursive programs to be computed efficiently by iterative imperative programs (with either an associative array or just with a finite fixed size static memory).

Acknowledgment

The author thanks the anonymous reviewers and Dr. Daniel De Carvalho for significant comments both in the content and in the language of the article.

References

[1] G. Berry. Bottom-up computation of recursive programs. *Theor. Inf. Appl.*, 10(3):47–82, 1976.

[2] R. S. Bird. Zippy tabulations of recursive functions. In P. Audebaud and C. Paulin-Mohring, editors, *Proc. of 9th Int. Conf. on Mathematics of Program Construction, MPC 2008*, volume 5133 of *Lecture Notes in Computer Science*, pages 92–109. Springer, 2008.

[3] J. Cowles and R. Gamboa. Contributions to the theory of tail recursive functions. 2004. Available at http://www.cs.uwyo.edu/~ruben/static/pdf/tailrec.pdf.

[4] D.E. Knuth. Textbook examples of recursion. arXiv preprint cs/9301113, 1991. Available at https://arxiv.org/abs/cs/9301113.

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

[6] M.S. Paterson and C.T. Hewitt. Comperative schematology. In *Proc. of ACM Conf. on Concurrent Systems and Parallel Computation*, pages 119–127. ACM, 1970.

[7] N.V. Shilov. Etude on recursion elimination. *Modeling and Analysis of Information Systems*, 25(5):549–560, 2018.