Adaptive Languages and a new Programming Style

APARECIDO VALDEMIR DE FREITAS [1,2] , JOÃO JOSÉ NETO [1]

[1] Escola Politécnica da Universidade de São Paulo
Depto. de Engenharia de Computação e Sistemas Digitais
Av. Prof. Luciano Gualberto, trav. 3, N^o 158.
Cidade Universitária – São Paulo - Brasil

[2] Universidade Municipal de São Caetano do Sul
Instituto Municipal de Ensino Superior de São Caetano do Sul
Av. Goiás N^o 3400 – São Caetano do Sul – CEP 09550-051 SP – Brasil
avfreitas@imes.edu.br and joao.jose@poli.usp.br

Abstract: - A programming style can be seen as a particular model of shaping thought or a special way of codifying language to solve a problem. Adaptive languages have the basic feature of allowing the expression of programs which self-modifying through adaptive actions at runtime. The conception of such languages calls for a new programming style, since the application of adaptive technology in the field of programming languages suggests a new way of thinking. With the adaptive style, programming language codes can be structured in such a way that the codified program therein modifies or adapts itself towards the needs of the problem. The adaptive programming style may be a feasible alternate way to obtain self-modifying consistent codes, which allow its use in modern applications for self-modifying code.

Key-Words: - Adaptive Devices, Adaptive Programming Languages, Adaptive Programming Style.

1 Introduction

The essential concept characterizing an adaptive device is its capacity to perform adaptive actions, which can be understood as procedure calls, built in the device and activated in reply to detected situations requiring behavioral changes of it [1].

Described by a finite and well-defined set of rules, such devices start operation at some pre-set initial configuration. The clauses forming this set of rules test the device current configuration and determine its new configuration.

An adaptive device has a subjacent formalism, e.g. an automaton, a grammar, a decision tree, etc. and an attached adaptive mechanism that allows the subjacent formalism to be dynamically changed [5].

Adaptive programming languages are adaptive devices that use a conventional programming language as its subjacent formalism.

The adaptive mechanism associated to a programming language gives it the self-modification feature.

Self-modifying programs have been used extensively at the early years of Computation, motivated by the lack of storage in ancient computers.

However, difficulties to read and maintain self-modifying code motivated the use of more reliable static-code-based avoiding self-modifications techniques instead [9].

Many recent works have been published in which self-modifying code is used in a new way. Among them, we may list: code protection, program compression, protection against undesired reverse engineering and code optimization [10] [11] [12] [13].

The self-modifying codes employed in machine languages are usually hard to write and to keep [16]. The technique we advance, however, employs the adaptive technology considering the use of adaptive functions specified by fixed and well-defined rules and can thus assure greater usability to the solution proposed.

With the adaptive programming style, general programs codified in programming languages can be conveniently structured so that the code can change or adapt itself towards the specific problem.

The adaptive programming style automatically leads programmers to non-conventional reasoning since a self-modifying behavior is promoted by adaptive actions.

Such a characteristic is typical of adaptive languages and demands special attention of the developers, who must anticipate the effects of adaptations on the executing code.

In this paper a new programming style based on code adaptivity is suggested.

Such a style will allow developing projects to use a self-modifying code in a consistent and disciplined way.

2 Adaptive Programming Style

Paradigms describe theories and procedures that, when used together, can represent a way of organizing knowledge.

There is a natural learning in accepting the paradigms that match our way of thinking and, as a matter of course, in rejecting new, different paradigms, or – in some way – models that clash with the way we think. The more attached we are to a given way of thinking, the more resistance we will oppose to any evidence or arguments against it.

Among the different ways of making a paradigm or way of thinking known, language assuredly counts amid the most representative.

The language through which thought is expressed essentially reflects the idea's nature. The association between thought and language becomes even more critical when we extend the concept to the field of programming languages. So, the language used by a programmer in solving a problem is closely related to his way of thinking or implementing the solution to a problem [2].

When the target described by a paradigm contains the items and relationships present in the problem's field of interest, the task of modeling a solution in that field is considerably simplified.

For example: the logical paradigm tends to materialize the solution to a problem by composing the predicates and relations, while the functional paradigm focuses use and composition of functions.

If we are tackling a programming language which allows direct representation of array models, the solution to the problem involving matrix arithmetic will be much simpler.

On the other hand, if the programming language to be used doesn't directly abstract the problem's field entities (here, the matrices), the solution must be attained through simulation of such entities with the help of elements available in the language. This task makes it harder to implement the process and reduces the solution's degree of expression since the programmer's way of thinking will not be directly mapped in the elements of the implementation language [2].

After these considerations, we can infer that programming languages involving written solutions to the problems direct the programmer's mind so much that he will deal with the problem according to the view the language paradigm imposed on him.

It is quite common to run across programmers dealing for so long with a given programming language the paradigm of which has so strongly associated with their way of thinking that the solutions they come up with are invariably modeled on the constructions available in the language paradigm. These programmers most often have a hard time struggling to break their usual paradigm and start employing other programming paradigms.

The new reasoning form required by adaptive technology and self-modifying programming motivates a new programming style, since the behavior of programs written in adaptive languages depends on the adaptive functions stating the way code is dynamically modified.

In non-adaptive programs, the static code is never modified in runtime, allowing traditional and well-accepted software engineering methodologies to be applied.

In adaptive programming, however, the several runtime instances of the running code, generated by the adaptive functions, must be considered.

Therefore, adaptive programming requires non-conventional programming methods, which is a subject to be further investigated.

3 Adaptive Languages

Adaptive technology refers to the techniques and methods associated with the adaptive devices' practical applications. Chronologically, such devices came about along researches in the field of formal languages and automata [3] [4].

The formalism, however, lays open instances which may apply to several other fields [5].

Formalisms based on state machines are tools often used to describe and model systems. At each operation stage in these machines, the devices take some configuration representing the whole lot of information stored so far by the machine [1].

Programming adaptive languages are adaptive devices using a conventional programming language as an underlying mechanism. As their execution runs, the program written in an adaptive language will uncover a self-modifying behavior by activating adaptive actions.

Adaptive actions represent the execution of adaptive functions. When such functions are well-designed, they allow self-modifying code to be used in a better way.

While executing an adaptive function in some instant t_i, a particular code instance C_i is passed to a function which generates another code instance C_{i+1} by executing elementary adaptive actions responsible for applying editing primitives to the code being executed.

We may define an adaptive function F on self-modifying code through the following application:

$$F : D \rightarrow D$$

where D represents the set of host language code used as the subjacent formalism of the adaptive language.

Adaptive function F acts on a domain representing the set of codes in the subjacent language.

The subjacent language may be selected from any paradigm. In this paper, we have chosen a Lisp dialect only for experimental purposes, without any explicit preference for the functional paradigm [6].

That being said, we will set out from a functional nucleus based on untyped lambda calculus which will act as an interpreter of the language the user will codify his programs with [7].

Upon this basic functional nucleus, we will project an extensible adaptive layer which will evaluate adaptive action calls responsible for the code self-modification.

Adaptive formalisms embodied in already existing programming languages (here, a functional language) will display, at first, a code block which may be directly processed by the basic functional nucleus interpreter until the execution of some specified adaptive action in the program represented in this code block takes place.

By processing the adaptive actions, a new instance of the program is reached (in functional language, we mean) and the operation is once more switched to the functional nucleus, which will take the operation on.

To process our functional adaptive language, there is required a processing environment made up by a functional nucleus and a control module – represented by the adaptive machine – to which will go the responsibility of managing the operation of the self-modifying codes written in this language.

Thus, the functional nucleus will look like the classical functional languages Lisp, which renders unnecessary the formal specification of host language (either syntactic or semantic), since the adaptive functions will be defined by elements made available by classical functional languages adopted as underlying mechanism.

As the underlying language used in this study is based on expressions, any program written in this language may be reduced to a single expression (the first element denoting a function, which can be composed (nested) from several other expressions (native or codified by the user).

In order that the adaptive functions may produce the code self-modification, we must somehow address the expression of the source program undergoing adaptiveness and change them by means of the elementary action calls existing in the adaptive layer. Such elementary actions, at runtime, will perform the inclusion, exclusion or alteration of the expressions according to convenience to the particular problem at issue.

Viewing that the expressions corresponding to the codes of programs written in underlying language show the tree structure, it is possible to identify every node of the tree with the respective opening of brackets – which stand for calls of components functions – through labels that enable to carry out the references.

In the project of adaptive functions it is enough to link the labels to the expressions actually taking part in the code self-modification.

It becomes then possible – by means of labels associated with the several functions (native or defined by the user) composing the user's code – to design the adaptive functions – responsible for the self-modification of the program – the behavior of which will be similar to the classical procedures of tree edition.

Therefore, in a way analogous to the processing of nodes in a tree, in our adaptive language model, the adaptive functions will establish a "string-processing" in the lambda expression corresponding to the program, creating a new string (or a new lambda expression) adhering to the labels addressed by the adaptive actions.

4 Adaptive Expressions

Adaptive expressions are common expressions built in the host language displaying calls of adaptive functions (adaptive actions) with the responsibility to implement the self-modifying behavior typical of adaptive languages.

By composing the library functions present in the device adaptive layer, it is possible to define the set of elementary actions responsible for the consultation, addition and elimination of the underlying language functions, which, at runtime, will promote the code adaptivity.

To exemplify the use of adaptive functions, let us consider the calculation of nth in the Fibonacci sequence.

Fibonacci numbers are defined recursively by the equations: $F_1 = 1$, $F_2 = 1$, $F_i = F_{i-1} + F_{i-2}$, $i > 2$.

There follows a functional solution to the Fibonacci series:

```
(define (fib n)
                ( if (<= n 2)
                        1
                       ( + (fib (- n 2) ) (fib (- n 1)))
                )
        )
```

Let us now observe the solution to a problem using the adaptive style. The adaptive language will be processed by an adaptive environment requiring from the user the source text corresponding to the program's initial instance and its parameters.

This source text will be saved in an area of the adaptive layer we will call *buffer*. The area relative to the parameters will be called *param*.

Thus, at the time of starting the adaptive code, the following save operation will be processed.

```
    (set  'buffer (load "fib_adapt.lsp"))
   (set  'param (load "param.lsp"))
```

At the beginning, the adaptive environment will interpret the initial code instance, generating an equivalent in lambda expression and updating the *buffer* variable.

For instance, to make the calculation for the Fibonacci number for value 10, the underlying language interpreter will evaluate:

```
> (eval buffer 10)
55
```

To reason adaptively, we can consider our initial source program (*fib*, for our purposes) as a function capable of calculating, in a simple way, the Fibonacci number for values 1 and 2. For values over 2, the *fib* function requires adaptation.

The initial code, then, must be conveniently modified by a call of adaptive function (here named *f_adapt1*), which will account for the instance self-modification of the initial code.

An analogy can be drawn in this case between the adaptive style and the Mathematical Induction Method, which – based on the Finite Induction Principle – is often turned to demonstrate the facts referring to infinite sequences.

Using such analogy, it is possible to associate the induction basis with our initial version of the code and the inductive step with the successive adaptive function calls at runtime, as shown in Fig-1.

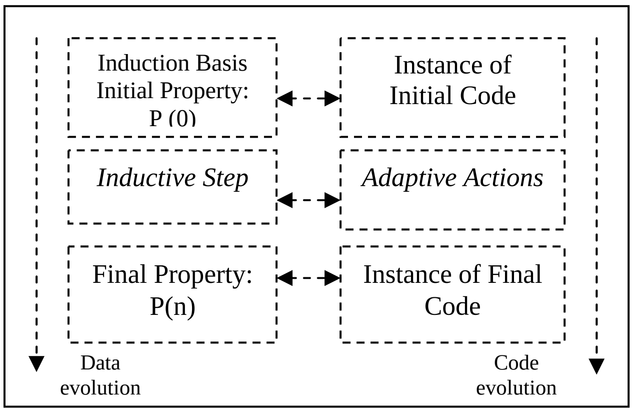
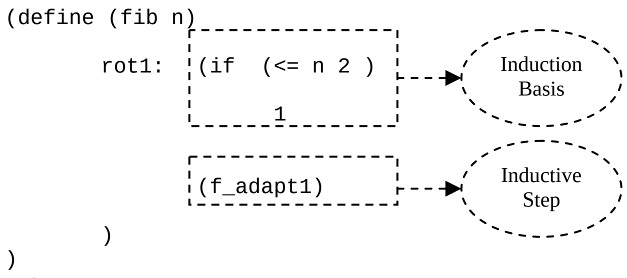


Fig-1 – Analogy between Inductive Step and Adaptive Actions

These considerations made and by naming the adaptive function *f_adapt1*, the initial code, in the light of the adaptive style, will be defined as:



```
(define (fib n)
        rot1:  (if  (<= n 2 )
                     1

               (f_adapt1)
        )
)
```

After the translation as lambda expression, the adaptive device will relay the control to the underlying language interpreter by means of function:

```
(eval buffer <param>)
```

The initial code will thus be conveniently modified by the function call adaptive function *f_adapt1*, responsible for the self-modification of successive code instances, the project of which will be revealed further ahead.

Considering that the image of the initial code is saved in the *buffer* variable of the adaptive layer, this image must be modified by the adaptive call (in which the code self-modification will take place).

The function *f_adapt1* will process the switching of context from the underlying layer to the adaptive layer.

Our adaptive function can be specified by the following rules:

;; adaptive code
(define (f_adapt1)
        (**insert_after** func rot1) ;;node insertion rule
        (**delete** rot1) ;;node delete rule
)

The adaptive machine will process the adaptive function *f_adapt1* so as to insert *func* function after *rot1* label in order to eliminate *rot1* as the next step (*rot1* corresponding to *if* expression in the initial code instance).

The *func* function may be represented by the code:

```
        (define ( func n)
                ( + (fib (- n 2) ) (fib (- n 1))
        )
```

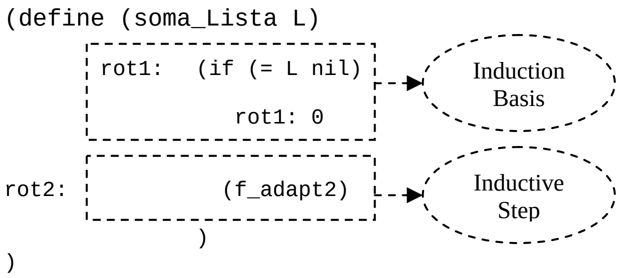
The adaptive machine will call the underlying interpreter to interpret *func* and will save the corresponding lambda expression in a work field within the adaptive environment.

After the processing of adaptive action *f_adapt1*, a second instance of the source code is then created and the control returns to the underlying interpreter to carry the program operation on.

Here is another example to illustrate the concept of adaptive expressions. Let us consider the function *add_list* which returns the sum of elements on a list.

At each recursive instance of function add_list, a new part of the sum is created, without – therefore – any communication of data among the instances (only the value of the sum being returned), which proves the non-existence of side effects.

The solution adopted to the problem of the sum of the elements on a list is typically functional. It is time to consider the application of the adaptive style to the problem. The initial code can be defined by:



```
(define (soma_Lista L)
        rot1:   (if (= L nil)
                   rot1: 0
rot2:                (f_adapt2)
                  )
)
```

The initial instance is capable of returning the sum of the elements on the list, according to the code above, only if this is empty, obviously returning to value zero.

If the list passed as parameter is not empty, the adaptive function *f_adapt2* will be called in order to modify the initial code, represented by a lambda expression. In this case, everything takes place as if the code had to learn “something new” or to “adapt itself” to deal with parameters of non-empty lists.

Viewing that, in the adaptive solution, the initial instance is restricted to a limited situation (empty list), its contents will be represented by fewer code lines than the functional solution defined previously.

The adaptivity of the solution, therefore, will be materialized as soon as the function *f_adapt2* is called.

While no adaptive call is effected, the processing of the underlying language will follow the ordinary way corresponding to the execution of an usual functional language.

The following may be the contents of our adaptive function *f_adapt2*:

```
 (define (f_adapt2)
          (insert_after  '( (setq total) ( +  (first L )
                   (soma_lista (rest L) ) ) rot1 )
           (delete rot2)
)
```

As the result of the application of adaptive function *f_adapt2* on the instance of initial code, a new instance of code C_2 will be created and reflected on the contents of the *buffer* variable under the control of the adaptive device.

The adaptive machine, after generating the new instance C_2 (the result of the application of adaptive function *f_adapt2*), will return the contents of the updated *buffer* by means of the adaptive action and take the program operation on.

5 Further Considerations

Self-modifying codes, especially those used in low-level languages, are generally hard to write, document and keep.

Our proposal, however, turns on the use of adaptive technology, which employs adaptive functions to this end. These are designed according to well-established rules that, if applied carefully, insure greater usage capacity to the suggested mechanism.

As a rule, the program is divided into blocks, in a top-down methodology. With the adaptive languages, a program may be structured through bottom-up techniques, with the program modifying towards the specific code that solves the problem.

Since the code undergoes self-modification with the adaptive languages and as a matter of course there is a possibility of a burst in code space and time, it is wise to set limits to the operation executed by the adaptive actions, so that alterations may be predicted and controlled.

To reach this goal, it is highly recommended to develop a study determining the growth rate of the code in order to measure the code self-modification resulting from the successive application of adaptive actions.

Such limits, along with the respective analytic functions of code and time growth, must constitute a part of a design method of adaptive functions to be developed in future articles.

To make the project of adaptive functions lighter for the designer, the adaptive environment is required to supply tools that refine the dynamic code generated in this kind of programming, such tools having to control all the code regions subject to alterations, pointing out in a likewise dynamic way the parts of the program liable to adaptivity.

6 Conclusions

The examples included in this article show that the use of adaptive languages entails the development of a particular way of thinking and reasoning – natural and spontaneous – as a consequence of the incorporation of the self-modifying mechanisms present to the adaptive technology into the code.

This way of thinking implies envisaging a new programming style, since the behavior of the adaptive devices is directly associated with a set of rules that defines it, and undergoes changes as the code evolves.

Such feature, typical of the adaptive devices, requires special care from anyone developing applications of this kind, such as a reasoning style and a programming discipline capable of predicting the effects of the adaptive actions on the behavior of the device.

In the case of adaptive languages in particular, the first step to follow is to analyze meticulously the first instance of the program comprising the written code in the adaptive language. In other words, it takes a lot of attention to define the adaptive program's induction basis.

Among the several functions comprising the instance of the initial code, those likely to take a role in some adaptive action must be selected and linked to some form of reference (necessarily present in the underlying language), applying, for example, the mechanism of labels illustrated in this article.

From this study, it is likely to result the inductive step of the adaptive program project, a complement to the process of self-modifying programs project put forth in this article.

This paper has shown that adaptive techniques may be a feasible way to build consistent self-modifying programs provided that some discipline is imposed to the use of language's adaptive mechanism.

There are many subjects that require further improvements, such as: a more detailed method for designing and checking reliable self-modifying code.

Tools and environments must also be developed to support self-modifying programs to be developed and debugged.

Another target for a near future is to design and implement a programming language with adaptive features which will give the programmers an adequate notation for expressing dynamically-changing programs in a reliable way.

Obviously, further research involving subjacent languages from different paradigms will be essential to build a well-founded technology based on programming language adaptivity.

References:

[1] Neto, João José - Adaptive Rule-Driven Devices - General Formulation and Case Study**.** Lecture Notes in Computer Science. Watson, B.W. and Wood, D. - Implementation and Application of Automata 6th International Conference, CIAA 2001, Vol.2494, Pretoria, South Africa, July 23-25, Springer-Verlag, 2001, pp. 234-250.

[2] Timothy A. Budd - Multiparadigm Programming in LEDA, Oregon State University, Addison-Wesley Publishing Company, Inc, 1995.

[3] Neto, João José - Contribuição à metodologia de construção de compiladores. São Paulo, 1993, 272p. Thesis (Livre-Docência), Escola Politécnica, Universidade de São Paulo. (in Portuguese)

[4] Neto, João José - Adaptive Automata for Context -Sensitive Languages. SIGPLAN NOTICES, Vol. 29, n. 9, pp. 115-124, September, 1994.

[5] Pistori, Hemerson – Tecnologia em Engenharia da Computação: Estado da Arte e Aplicações. Tese de Doutorado – Escola Politécnica da Universidade de São Paulo – 2003. (in Portuguese)

[6] Rocha, Ricardo Luis de Azevedo da e Neto, João José - Uma proposta de linguagem de programação funcional com características adaptativas. IX Congreso Argentino de Ciencias de la Computación, Argentina, 6-10 Outubro - 2003. (in Portuguese)

[7] Freitas, A. V. - Neto, João José – Adaptive Device with underlying mechanism defined by a programming language - 4th WSEAS International Conference on Information Security, Communications and Computers (ISCOCO 2005).

[8] Barendregt, H.P. - The Lambda Calculus: its syntax and semantics – (2nd ed.), North-Holland, 1984.

[9] Philip K. McKinley, Seyed Masoud, Sadjadi, Eric P. Kasten, Betty H. C. Cheng – Composing Adaptive Software - Michigan State University - IEEE Computer Society – 2004.

[10] Anckaert B., Madou M. and Bosschere K.D. – A model for Self-Modifying Code - Proceedings of the 8th Information Hiding Conference, 10-12 July 2006.

[11] Madou M., Anckaert B., Moseley P. Debray S., Sutter B. D., Bosschere K. – Software Protection through Dynamic Code Mutation – Conference International Workshop on Information Security Applications – WISA 2005, LNCS 3786 pp 194-206.

[12] Yamamoto L., Tshudin C. – Harnessing Self-Modifying Code for Resilient Software – Proc. 2nd IEEE Workshop on Radical Agent Concepts – 2005

[13] Giffin J.T., Christodevescu L.K., Strengthening Software Self-Checksumming via Self-Modufying Code – 21st Annual Computer Security Applications Conference – December 5-9,2005 – Tucson, Arizona.