REFLECTIVE MONADIC ADAPTATION*

Ján KOLLÁR, Michal FORGÁČ
Department of Computers and Informatics, Faculty of Electrical Engineering and Informatics, Technical University of Košice,
Letná 9, 042 00 Košice, tel. 055/602 2576, E-mail: Jan.Kollar@tuke.sk, Michal.Forgac@tuke.sk

ABSTRACT

The purpose of software evolution is adaptation of a software system according to the new external or internal requirements unlike software maintenance, which purpose is mainly removal of some faults. Software evolution includes also language evolution. We can view on a software system as an integral combination of a program and a language for this program. Program can be viewed as a sequence of statements that are aimed to produce some result. The execution is done by a platform that interprets the program's sequence of statements. The new result of a computation can be achieved by transformation of a program, an interpreter or both. In this paper we present adaptive experiment in the field of adaptability with utilization of metaprogramming and reflection using functional programming with monads. They are useful interface between functional and imperative world of programming.

Keywords: *software evolution, metaprogramming, reflection, adaptability, functional programming, monads*

1. INTRODUCTION

Modification of complex software systems after their delivery means difficult process. The most often reasons for such modification are on the one hand detected faults, which have to be fixed or on the other hand requirements for new functionality, which systems have to include (e.g. replacement of a system from one computational environment into a new one). Such modifications require additional costs. Even implementation of required changes takes in some cases longer than implementation of the first operational software version. Thus there is significant demand for reaching optimal methods in order to achieve effective modification of software systems.

Under modification it is possible to mean software maintenance or software evolution. These terms are often used as synonyms. We incline rather to the claim (according to e.g. [7]), that these terms do not express similar meaning. The purpose of maintenance (e.g. perfective, corrective, adaptive, preventive) is mostly in removal of some faults, the purpose of evolution (e.g. static, dynamic, anticipated, unanticipated) is adaptation of a system according to the new external or internal requirements, thus the first delivery is only first step in continuous evolving process.

Evolution in general means, that something has changed for better way. According to Lehman the term E-type software [8] denotes programs that must be evolved because they operate in or address a problem or activity of the real world. Changes in the real world will affect software and require adaptations to it.

Software evolution [1,11] includes also language evolution as an actual issue. Some projects may fail not because of bugs in programs, but because of the lack of recognition of language issues. Thus according to [4] software is composed from a program and a language. In this approach is language implementation (e.g. interpreters, compilers, other language-dependent tools) regarded as a metalevel of a program.

There is general agreement for the set of several important challenges on software evolution presented in [10]. We suppose that mainly four ideas play a significant role for our research: formal support for evolution, evolution as a language construct, support for multi-language systems, and post-deployment runtime evolution. The first challenge is about formal methods, which need to embrace change and evolution as an essential fact of life. The second challenge states, that programming languages should provide more direct and explicit support for software evolution. The third challenge is about support for multi-language systems, and claims that software evolution techniques must provide more and better support for multi-language systems. The last challenge is about post-deployment runtime evolution and calls for an urgent need for proper support of runtime adaptations of systems while they are running, without pause or stop them.

This paper is structured as follows: section 2 presents basic principles of metaprogramming and reflection, which are basis of our research. Adaptive language implementation as our previous work is presented in section 3. Section 4 deals with main principles of monads, which are used in section 5 in our proposal of adaptive metalevel architecture. Finally, section 6 concludes this paper.

2. METAPROGRAMMING AND REFLECTION

Metaprogramming is about writing programs that represent and manipulate other programs or themselves, i.e. metaprograms are programs about programs [3]. The impact of metaprogramming is obvious in traditional development processes, by sorting existing programs as transformation processes with inputs and outputs.

Reflection is an entity's integral ability to represent, operate on and otherwise deal with itself in the same way that it represents, operates on, and deals with its primary subject matter [3]. A metalevel provides information about selected system and makes the software self-aware. A base level includes the application logic.

There are two aspects of reflection [2]: introspection and intercession. Introspection is the ability of a program to observe and therefore reason about its own state. Interces-

*THIS WORK WAS SUPPORTED BY VEGA GRANT NO. 1/4073/07 ASPECT-ORIENTED EVOLUTION OF COMPLEX SOFTWARE SYSTEMS

sion is the ability of a program to modify its own execution state or alter its own interpretation or meaning. Both aspects require a mechanism for encoding execution state as data, providing such an encoding is called reification [12]. With introspection it is possible to read and access reifications whereas intercession allows also modify these reifications.

Reflection can be divided into structural and behavioural reflection [14]. Structural reflection represents the ability of a program to access a representation of its structure, as it is defined in the programming language. For instance, in an object-oriented language, structural reflection gives access to the classes in the program as well as their defined members. Behavioural reflection represents the ability of a program to access a dynamic representation of itself, that is to say, of the operational execution of the program as it is defined by the programming language implementation (processor). In an object-oriented language, behavioural reflection could for instance give access to base-level operations such as method calls, field accesses, as well as the state of the execution stack.

A programming language is said to be reflective if it provides an explicit representation (i.e. reification) of entities that either represent program building blocks (e.g. classes, methods) or are involved in program execution (e.g. stack, garbage collector) [14]. Developers thus can define system (software) functionalities and also new program building blocks or execution mechanisms (how functionalities will be performed). From the object-oriented point of view, objects that define program functionalities are called base level objects or base-objects, objects defining program building blocks or execution mechanisms are called metalevel objects or meta-objects [3].

Meta-object protocol (MOP) [14] offers possibility for extension of a programming language and adapts respective execution mechanisms. Using a reflective language it is possible to implement both objects and metaobjects.

3. ADAPTIVE LANGUAGE IMPLEMENTATION

A programming language is a medium to express computation, which is defined by its syntax and its semantics. An implementation of a programming language is the realization of its syntax and its semantics, which comprises a translator and a run-time system [9].

The result of a computation depends on both a program P and an interpreter I. There are two general ways, how to change this result. The first approach is based on transformation of a program (an interpreter will stay unchanged), the second approach is based on transformation of an interpreter (a program will stay unchanged). There is another possible way, which represents combination of previous ways (both a program and an interpreter will be changed).

Our approach is related to the second group of transformation because it is based on non-program transformation. Our idea of interpreter transformation is widen by transformation of various components of the execution mechanism. Thus, according to our approach, execution is a synonym for transformation in general, such as translation, type checking, code generation, loading, interpretation, modelling, algebraic specification, and even for informal but constructive thinking about algorithmic problems.

Foundations of our adaptive language implementation were presented in [6], where we have introduced simple LL(1) language and its adaptive interpreter which consists of lexical analyser, adaptive translator and evaluator (every module was implemented in Haskell functional language) such that interpreter k = eval . translate k . lexical, where k is a variant, which depends on the result in the stack (in our case, the result was a number value). This means, that depending on the result of interpretation, the LL(1) language should be changed, and the next interpretation follows different semantics, i.e. potentially different result of the same source expression. This language had simple grammar of operations $(+)$, $(-)$, $(*)$, and $(/)$ with various associativity and priority. Then according to the result in the stack, the grammar of the language (associativity and priority of listed operations) was changed according to the metadata (which are represented by variants). The principle of adaptability is depicted on the Fig. 1.

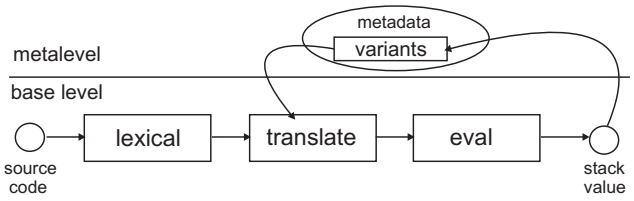

Fig. 1 Adaptive language implementation

4. MONADS

Functions in Haskell [5] are pure, meaning that they define a relationship between inputs and outputs, and have no side effects. Impure functions refer to external state, modify their behavior based on external state, or change external state.

Haskell makes impure functions available, but isolates them through the use of monads, which are actually built from pure functions. Monads [16] have two main purposes, they encapsulate common computational patterns and isolate code that interacts with the outside world. There are various types of monads, some of them are listed in the Table 1. Pure functions cannot call monad functions but monad functions can be called by other monad functions.

Table 1 Selected types of monads

Monad	Computation
Maybe	may not return a result
Error	can fail or throw exceptions
List	can return multiple results
IO	perform I/O
State	maintain state

A type m is a monad if it implements four operations [15]: bind (the $\gg=$ operator), then (the $\gg$ operator), return, and fail. On the Fig. 2 is depicted an example of a return function and on the Fig. 3 is depicted an example of a bind function for the IO monad.

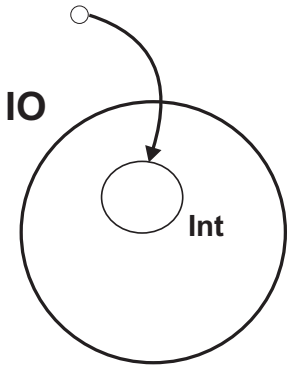

Fig. 2 Example of a return function for the IO monad

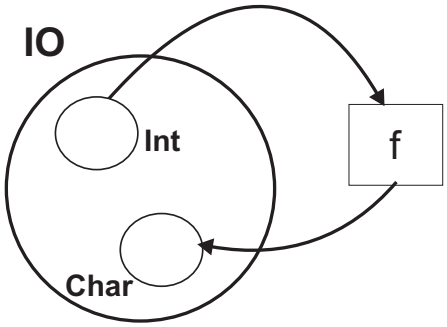

Fig. 3 Example of a bind function for the IO monad

Monads are a higher order type. Thus a monad m is something that acts as a container over some other type. A monad function is simply a function that returns a monad value. It can also take monad arguments. Developing hierarchical structure of monads seems to be promising approach for exploiting monadic style for adapting monadic systems.

5. EXPERIMENT WITH MONADS

The IO monad is one of the most frequently used monads. It is associated with input/output and acts as a container that isolates external side effects in a Haskell program. For example IO String is the type of value returned by user input funcions or IO Int can by returned by various random-number generators.

Our adaptive metalevel architecture (Fig. 4) requires input and output operations, because it works with files, thus the IO monad is appropriate solution for this purpose. This architecture consists of two layers: a base level and a metalevel. The base level consists of data files, lexical decomposition module and data generation module. The metalevel consists of various adapters and meta-transformers. This architecture is based on basic principles of metaprogramming and reflection.

Metalevel components access to the base level components - this operation's name is introspection, and then they modify or manage base level components, in our case files (if it is required) are influenced by adapters and the rest two modules are influenced by meta-transformers. Listed influences are operations, which name is intercession. Metalevel components may comunicate also with another metalevel components. From metaobject protocol point of view we can regard these components as metaobjects.

In our architecture, base level data is represented through 3 files:

1. source file (srcF), which consists of digits in String representation,
2. intermediate (data) file (inmF), which consists of list of integers [Int],
3. target (value) file (valF), which consists of integer value Int.

There are two types of transformers:

1. lexical decomposition transformer (lexT), which allows transformation srcF $\rightarrow$ inmF,
2. value generation transformer (genT), which allows transformation inmF $\rightarrow$ valF.

These two types of transformers can perform three types of transformations:

1. transformation$_{12}$ :: String $\rightarrow$ [Int], it is direct transformation,
2. transformation$_{23}$ :: [Int] $\rightarrow$ Int, it is direct transformation,
3. transformation$_{13}$ :: String $\rightarrow$ Int, it is strided transformation.

These transformations are controlled from metalevel through corresponding meta-transformers. There are six types of adapters on metalevel:

1. self_adapter$_{11}$ :: String $\rightarrow$ String,
2. self_adapter$_{22}$ :: [Int] $\rightarrow$ [Int],
3. self_adapter$_{33}$ :: Int $\rightarrow$ Int,
4. backward_adapter$_{21}$ :: [Int] $\rightarrow$ String,
5. backward_adapter$_{32}$ :: Int $\rightarrow$ [Int],
6. backward_adapter$_{31}$ :: Int $\rightarrow$ String.

If strided transformation is used, then it does not allow utilization of backward_adapter$_{21}$ and self_adapter$_{22}$.

Adaptability is performed according to given conditions which are in individual adapters. We can see these conditions as some variants. For example, variants for adapter$_{11}$ may depend on the lenght of a string (or on occurence of individual character in any position in the string), variants for adapter$_{21}$ or adapter$_{22}$ may depend on number of digits in the list (or on occurence of individual digit in any position in the list) and variants for adapter$_{33}$, adapter$_{32}$ or adapter$_{31}$ may depend on the final result calculated from the list of integers.

When an adapter finish changes, it calls metatransformer, which informs the transformer on the base level, that it can load input file and make transformation. Transformation process (with display possibility) for lexical decomposition module is as follows: readFile "srcF" $\gg=$ return.lexT $\gg=$ return.show $\gg=$ putStr, transformation process for value generation module (also with display possibility) is as follows:
readIntFile "inmF" $\gg=$ return.genT $\gg=$ return.show $\gg=$ putStr.

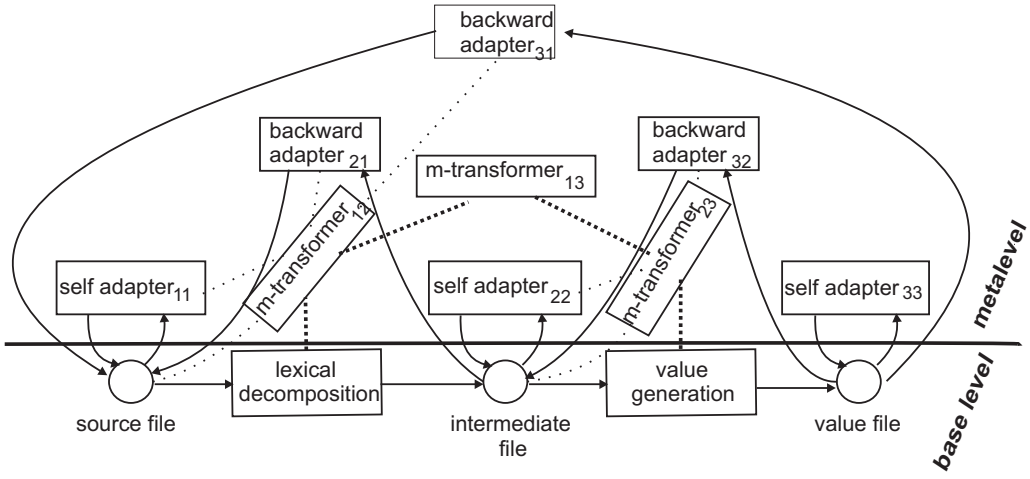

Fig. 4 Adaptive metalevel architecture

6. CONCLUSION

Presented adaptive experiment is our another contribution in the field of adaptability with utilization of metaprogramming and reflection. It utilizes monad principle, which constitutes an interface between functional and imperative world of programming.

The next step of our work will be proposal and implementation (or modification) of an existing execution environment, which will support language transformation (in sense of interpreter transformation). One of the suitable candidates for this purpose should be Smalltalk-like environment Squeak [13], which offers several reflective possibilities.

Practical utilization of our proposal can be evolution of complex software system through incremental design of its programming language with possibility of its modification (special importance for our research has mainly run-time modification). Another important issue is addition of new domain oriented languages during evolution of a software system. We have named process of language transformation as transformation of language through weaving (e. g. grammar weaving) as an analogy to the program weaving in aspect-oriented programming paradigm.

ACKNOWLEDGEMENT

This work was supported by VEGA Grant No. 1/4073/07 Aspect-oriented Evolution of Complex Software Systems.

REFERENCES

[1] BENNETT, K. – RAJLICH, V.: *Software Maintenance and Evolution: A Roadmap*, in A. Finkelstein, ed., The Future of Software Engineering, In A Finkelstein (ed.) The Future of Software Engineering, ACM Press, 2000, pp. 73–90

[2] BOBROW, I. D. G. – GABRIEL, R. G. – WHITE, J.L.: *CLOS in Context - The Shape of the Design Space*, In Object Oriented Programming - The CLOS Perspective. MIT Press, 1993, pp. 29–61

[3] CZARNECKI, K. – EISENECKER, U.: *Generative Programming: Methods, Tools, and Applications*, Addison Wesley (2005), 832 pp.

[4] FAVRE, J. M.: *Languages evolve too - Changing the Software Time Scale*, Eighth International Workshop on Principles of Software Evolution (IWPSE'05), 2005, pp. 33-44

[5] Haskell home page, http://www.haskell.org, 30. 4. 2008

[6] KOLLÁR, J. – PORUBÄN, J. – VÁCLAVÍK, P. – BANDÁKOVÁ – J., FORGÁČ, M.: *Functional Approach to the Adaptation of Languages instead of Software Systems*, COMSiS - Computer Science and Information Systems, 4, 2, 2007, pp. 115-129, ISSN 1820-0214

[7] LEHMAN, M. M.: *Laws of Software Evolution Revisited*, EWSPT96, Oct. 1996, LNCS 1149, Springer Verlag, 1997, pp. 108-124.

[8] LEHMAN, M. M. – RAMIL, J. F. – KAHEN, G.: *Replacement Decisions for E-type Software - Some Elements*, ICSE 2000 2nd Workshop on Economics-Driven Software Engineering Research, Limerick, Ireland, 6 Jun. 2000

[9] MALENFANT, J. – JACQUES, M. – DEMERS, F.: *A Tutorial on Behavioral Reflection and its Implementa-*

tion, Proceedings of Reflection 96, San Francisco, 1-20 (1996)

[10] MENS, T. – WERMELINGER, M. – DUCASSE, S. – DEMEYER, S. – HIRSCHFELD, R. – JAZAYERI, M.: *Challenges in Software Evolution*, In Proceedings of the Eighth International Workshop on Principles of Software Evolution (IWPSE'05), 2005, pp. 13-22

[11] ORIOL, M.: *An Approach to the Dynamic Evolution of Software Systems*, PhD Thesis, University of Geneva, Geneva, Switzerland, April 2004, 191 pp.

[12] RIVARD, F.: *Smalltalk: a Reflective Language*, Proceedings of Reflection '96 Edited by G. Kiczales San Francisco, April 1996, pp. 21-38

[13] Squeak homepage, `http://www.squeak.org`, 13. 5. 2009

[14] TANTER, E.: *From Metaobject Protocols to Versatile Kernels for Aspect-Oriented Programming*, PhD thesis, University of Nantes, France, and University of Chile, Chile. November 2004, 224 pp.

[15] TUROFF, A.: *Introduction to Haskell, Part 3: Monads*, `http://www.onlamp.com/pub/a/onlamp/2007/08/02/introduction-to-haskell-pure-functions.html?page=1`, O'Reilly, published in 08. 02. 2007

[16] WADLER, P.: *The essence of functional programming*, In 19th Annual Symposium on Principles of Programming Languages, Santa Fe, New Mexico, January 1992, pp. 1-14

Received Jun 9, 2009, accepted September 2, 2009

BIOGRAPHIES

Ján Kollár was born in 1954. He received his MSc. summa cum laude in 1978 and his PhD. in Computing Science in 1991. In 1978-1981, he was with the Institute of Electrical Machines in Košice. In 1982-1991, he was with the Institute of Computer Science at the University of P.J. Šafárik in Košice. Since 1992, he is with the Department of Computers and Informatics at the Technical University of Košice. In 1985, he spent 3 months in the Joint Institute of Nuclear Research in Dubna, Soviet Union. In 1990, he spent 2 month at the Department of Computer Science at Reading University, Great Britain. He was involved in the research projects dealing with the real-time systems, the design of (micro) programming languages, image processing and remote sensing, the dataflow systems, and the implementation of functional programming languages. Currently the subject of his research are adaptive languages and software systems.

Michal Forgáč was born in 1983. In 2006 he graduated at Technical University of Košice. He is working on his PhD. degree at the Department of Computers and Informatics, Faculty of Electrical Engineering and Informatics, Technical University of Košice. His scientific research is focused on the aspect-oriented programming paradigm, software evolution and adaptiveness of complex software systems.