Building Adaptive Language Systems*

JÁN KOLLÁR[1]
JAROSLAV PORUBÄN[1]

Technical University of Košice
Department of Computers and Informatics
Letná 9, 042 00 Košice, Slovakia
[1](Jan.Kollar,Jaroslav.Poruban)@tuke.sk

Abstract. The notion of change as a first-class entity in the language is the idea of software language engineering. Multiple metalevel concept is an essential demand for a systematic language approach, to build up adaptable software systems dynamically, i.e. to evolve them. A feedback reflection loop from data to code through metalevel data is the basic implementation requirement and the proposition for semi-automatic evolution of software systems. In this paper, we illustrate the ability for extensions primarily in horizontal but also in vertical direction of an adaptive system. From the viewpoint of adaptability, we classify software systems as being nonreflexive, introspective and adaptive. Introducing a simple example of LL(1) languages for expressions, we present its nonreflexive and adaptive implementation using Haskell functional language.

Keywords: Adaptive systems evolution, adaptive languages, semantic transformation, aspect oriented languages, domain specific languages, reflection.

(Received August 03, 2007 / Accepted February 02, 2008)

1 Introduction

Adaptability is mostly related to the area of software engineering – object oriented programming, aspect-oriented programming, intentional programming, template programming, etc. concentrating on transformation of program codes. As a very interesting research direction, programming (or even modelling) languages should provide more direct and explicit support for software evolution. The idea would be to treat the notion of change as a first-class entity in the language. This is likely to cause a programming paradigm shift similar to the one that was encountered with the introduction of object-oriented programming.

Our work comes out from three areas: from the area of monadic purely functional languages, from our research on process functional language – an environment oriented language without assignments [11, 12, 13, 14, 15], and finally, from the area of aspect oriented programming, dealing with the problem of modularizing crosscutting concerns.

It may be noticed, that the frequently asked question *What an aspect is* [21] is irrelevant, if we have an adaptive aspect oriented language, which unfortunately has not been constructed, otherwise such language would be simply adapted on a new, just arising aspect.

That is why we oriented our work to the analysis of the principles of adaptiveness, renewing it by exploiting at the area of computer languages rather than software engineering concepts.

Below we mention some essential concepts related to adaptive software evolution.

Metaprogramming is about writing programs that represent and manipulate other programs or themselves [4].

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 deal with its pri-

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

mary subject matter. Reflection is a fundamental concept of self-adaptive systems.

The main idea of applying reflection as a general principle for flexible systems in software engineering is to split a system into two parts: metalevel and a base level. 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: *introspection* and *intercession*. *Introspection* is the ability of a program to observe and therefore to reason about its own state. *Intercession* is a higher degree of reflection, since it 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*.

Different languages provide different levels of reflection. For example, the Java reflection API, allows a programmer to discover methods and attributes in classes at runtime, and to create objects of classes, whose names are not known until runtime. Similarly, it is possible to call methods and access attributes whose names are not known until runtime, because they may be discovered with the help of reflexive facilities, or they may be computed at runtime. Thus, Java's reflexive facilities primarily support introspection. In contrast to Smalltalk, Java does not allow to directly modify classes or methods by modifying their metaobjects at runtime, that is, it does not support intercession.

Metaobjects are objects that represent methods, execution stacks, the processor, and nearly all elements of the language and its execution environment. Most importantly, regular language code can access and modify these metaobjects.

There is a principal difference between metaclass and metaobject. Java metaclass is static data while metaobjects in Smaltalk is dynamic data. As we will see, dynamic metadata is the proposition for building adaptive systems.

In Section 2 we introduce our classification of software systems from the viewpoint of the degree of reflexive behavior, and we analyze three selected cases. We also present the conception of multilevel adaptive language system.

In Section 3 we present LL(1) language for expressions and its nonreflexive implementation. The crucial points of its adaptive version are presented in Section 4.

Our approach is functional, and we use Haskell – purely functional language and for practical experiments we have used Hugs98 system.

This allows us to express adaptive translator concisely and transparently.

Although the adaptation of the translator is not general, i.e. adapted rules are not produced as an instances of extended BNF form, and metalevel is scattered in multiple modules, the principle of adaptation is visible – it is generalization and abstraction.

We illustrate our approach just using one feedback loop (from interpreter to translator), but such loops may be formed between any subsequent phases.

2 Systems Behavior Classification

In this section we classify software systems from the viewpoint of their degree of adaptive behavior.

We recognize set $\mathbf{N}$ (of system elements) with *nonreflexive* behavior, set $\mathbf{I}$ with *introspective* behavior, and set $\mathbf{A}$ with *adaptive* behaviour.

Behaviorally, $\mathbf{I}$ is stronger than $\mathbf{N}$ (we write it $\mathbf{I} > \mathbf{N}$), since no element in $\mathbf{N}$ can behave in introspective manner, but all elements in $\mathbf{I}$ can behave in a nonreflexive manner. By an analogy, $\mathbf{A}$ is stronger than $\mathbf{I}$, so the relation for different degrees of an adaptive behavior is in (1).

$$\mathbf{N} < \mathbf{I} < \mathbf{A} \tag{1}$$

Provided that a system consists of sets of different behavior, then they are structurally disjunctive. This is inevitable proposition for the existence of feedback loops from code to the same code via metalevel data, as we see below.

We mention that the execution is a synonym for transformation in general, such as translation, type checking, code generation, loading, interpretation, modeling, algebraic specification, and even for informal but constructive thinking about algorithmic problems. This is so, because we relate our classification to dynamic transformations of any kind, and then there are no strict boundaries between different types of transformations.

2.1 Nonreflexive Execution

Machine code – the constant set ${}^0\mathbf{C}$ of instructions at base level 0 does not vary during execution, and then the execution changes data ${}^0\mathbf{D}^{(k)}$ (the set of data records on the stack or in the heap) to a new data set ${}^0\mathbf{D}^{(k+1)}$. An execution step is the transformation of configuration (2).

$${}^0\mathbf{C} \ \stackrel{\times}{\longrightarrow} \ {}^0\mathbf{D}^{(k)} \quad \Longrightarrow \quad {}^0\mathbf{C} \ \stackrel{\times}{\longrightarrow} \ {}^0\mathbf{D}^{(k+1)} \tag{2}$$

In (2), the relation $(\stackrel{\times}{\longrightarrow})$ denotes that many instructions from ${}^0\mathbf{C}$, can access many data records ${}^0\mathbf{D}$.

This execution is nonreflexive, because there is no feedback loop from data to code, and no possibility given to code to observe or even to change itself.

2.2 Introspective Execution

In an introspective execution, code ${}^0\mathbf{C}$ constructs and changes data ${}^0\mathbf{D}^{(k)}$, as in nonreflexive execution. In addition to this, each subset of the set ${}^0\mathbf{D}^{(k)}$ refer (which we designate by $\nwarrow -$) exactly one element of static data ${}^1\mathbf{S}$, and this data refers ($\nwarrow +$) a subset of code ${}^0\mathbf{C}$ according (3). Static data set ${}^1\mathbf{S}$ at level 1 are metalevel static data to the level 0.

$$\begin{array}{ccc} {}^1\mathbf{C} & \xrightarrow{\times} & {}^1\mathbf{S} \\ & +\swarrow \quad \nwarrow - & \\ {}^0\mathbf{C} & \xrightarrow{\times} & {}^0\mathbf{D}^{(k)} \end{array} \Longrightarrow \begin{array}{ccc} {}^1\mathbf{C} & \xrightarrow{\times} & {}^1\mathbf{S} \\ & +\swarrow \quad \nwarrow - & \\ {}^0\mathbf{C} & \xrightarrow{\times} & {}^0\mathbf{D}^{(k+1)} \end{array} \tag{3}$$

Since metalevel data (set of records) ${}^1\mathbf{S}$ is static, metacode ${}^1\mathbf{C}$ may produce it once, and then the execution of ${}^1\mathbf{C}$ is finished. Clearly, such metacode cannot be runtime process, and execution of ${}^0\mathbf{C}$ is nonadaptive. However, it is introspective, because of existence of feedback loop from code ${}^0\mathbf{C}$ to code ${}^0\mathbf{C}$ via data set ${}^0\mathbf{D}$ and some metadata element from ${}^1\mathbf{S}$.

2.3 Adaptive Execution

Adaptive execution step is defined in (4). In this case, metalevel data ${}^1\mathbf{D}^{(m)}$ can change in runtime to data ${}^1\mathbf{D}^{(m+1)}$, by execution of metalevel code ${}^1\mathbf{C}$. This code itself is nonreflexive, since there is no feedback loop via metadata at level 2. On the other hand, new ${}^1\mathbf{D}^{(m+1)}$ may result to new ${}^0\mathbf{C}^{(k)}$, continuing its execution at level 0.

$$\begin{array}{ccc} {}^1\mathbf{C} & \xrightarrow{\times} & {}^1\mathbf{D}^{(m)} \\ & +\swarrow \quad \nwarrow - & \\ {}^0\mathbf{C}^{(k)} & \xrightarrow{\times} & {}^0\mathbf{D}^{(k)} \end{array} \Longrightarrow \begin{array}{ccc} {}^1\mathbf{C} & \xrightarrow{\times} & {}^1\mathbf{D}^{(m+1)} \\ & +\swarrow \quad \nwarrow - & \\ {}^0\mathbf{C}^{(k+1)} & \xrightarrow{\times} & {}^0\mathbf{D}^{(k+1)} \end{array} \tag{4}$$

It means that code at level 0 may be not just introspective, but also adaptive, and this fact is essential for an adaptive execution.

It is easy to see, that level 2 may be built upto level 1 similarly as level 1 upto level 0, etc. Such chain of metalevels represent abstractions of previous level, and this abstraction we recognized optimal, provided that

1. Cardinality relation $|{}^{(\ell+1)}\mathbf{D}| \ll |{}^{\ell}\mathbf{D}|$ holds, since each metalevel should express its base level concisely, and
2. Computational time relation ${}^{(\ell+1)}\tau \ll {}^{\ell}\tau$ hold, which says that computational time at a metalevel should be significantly shorter than that at a base level.

In Section 3 we will introduce simple LL(1) language for expressions and then, in Section 4, we present our conception of dynamic adaptiveness, applied to this language.

3 Nonreflexive Language

Syntax of LL(1) language of expressions is written in extended BNF form (5).

$$\begin{array}{lcl} E & \rightarrow & A \,\{\, (\text{"}+\text{"} \mid \text{"}-\text{"})\; A \,\} \\ A & \rightarrow & B \,[\, (\text{"}*\text{"} \mid \text{"}/\text{"})\; A \,] \\ B & \rightarrow & \mathit{const} \mid \text{"("}\; E \;\text{")"} \end{array} \tag{5}$$

where $[\varphi] = (\varphi|\ \varepsilon)$, ε is empty symbol, φ is a syntactic expression, and $\{\varphi\} = \varepsilon \mid \varphi \mid \varphi\ \varphi \mid \ \ldots$ is the transitive closure.

The priority and associativity of operations defined by LL(1) grammar (5), is as follows: operations $(+)$ and $(-)$ are left-associative and they are on lower priority than operations $(*)$ and $(/)$ that are right-associative.

The nonreflexive implementation of this language is depicted in Fig. 1.

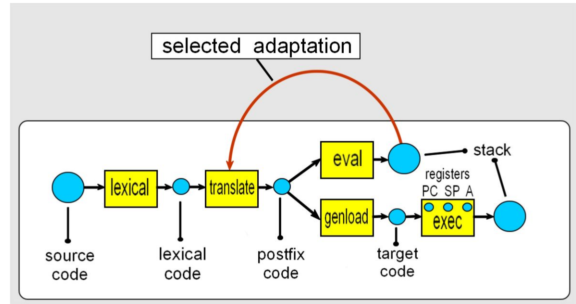

Figure 1: Nonreflexive Language

In addition, the task of adaptation, which we will solve in Section 4 is depicted in Fig. 1 by red arc, which forms essential feedback loop between the result of interpretation and the translation to postfix code.

Lexical analysis is defined by the translation schemes $\mathcal{L}$ (6) and implemented by function `lexical`.

$$\mathcal{L}[\![\, \text{"}+\text{"} \mid \text{"}-\text{"} \mid \text{"}*\text{"} \mid \text{"}/\text{"} \mid \text{"("} \mid \text{")"} \mid \text{const} \,]\!] = \mathbf{AL} \mid \mathbf{SL} \mid \mathbf{ML} \mid \mathbf{DL} \mid \mathbf{LL} \mid \mathbf{RL} \mid \mathbf{VL}\ (\text{aToI const}) \tag{6}$$

where const is string form of an integer, which is translated to value using aToI. We use sum operation |, instead of defining $\mathcal{L}$ for each symbol separately.

The semantics given by (7) yields the implementation by function `translate`, which translates lexical code (symbols) to postfix code.

$$
\begin{array}{l}
\mathcal{E}[\![\ A\ \{\ (\mathbf{AL} \mid \mathbf{SL})\ A'\ \}\ ]\!] = \\
\quad \mathcal{A}[\![\ A\ ]\!]\ \{\ \mathcal{A}[\![\ A'\ ]\!]\ (\mathbf{Add} \mid \mathbf{Sub})\ \} \\[1em]
\mathcal{A}[\![\ B\ [\ (\mathbf{ML} \mid \mathbf{DL})\ A\ ]\ ]\!] = \\
\quad \mathcal{B}[\![\ B\ ]\!]\ [\ \mathcal{A}[\![\ A\ ]\!]\ (\mathbf{Mul} \mid \mathbf{Div})\ ] \\[1em]
\mathcal{B}[\![\ \mathbf{VL}\ v \mid \mathbf{LL}\ E\ \mathbf{RL}\ ]\!] = \\
\quad \mathbf{Push}\ v \mid \mathcal{E}[\![\ E\ ]\!]
\end{array}
\tag{7}
$$

Translation starts by $\mathcal{E}[\![\ E\ ]\!]$, since E is starting symbol.

Interpretation is defined by function `eval`, which produces the result on the stack, when applied to postfix code according (8).

$$\texttt{eval}\ pcode \tag{8}$$

Code generation and loading are composed into single pass. The transformation of postfix code operations to machine code instructions is defined by scheme $\mathcal{C}$ in (9)

$$
\begin{array}{lcllcl}
\mathcal{C}[\![\mathbf{Add}]\!] & = & 1 & \mathcal{C}[\![\mathbf{Sub}]\!] & = & 2 \\
\mathcal{C}[\![\mathbf{Mul}]\!] & = & 3 & \mathcal{C}[\![\mathbf{Div}]\!] & = & 4 \\
\mathcal{C}[\![\mathbf{Push}\ x]\!] & = & 5\ x & & &
\end{array}
\tag{9}
$$

where x is an integer value, stored on the target code immediately after code 5 of instruction $\mathbf{Push}$.

Machine code is generated and produced to memory (represented as a list) by function `genload`, which performs code generation as well as loading actions and this function is invoked by the application (10)

$$\texttt{genload}\ pcode \tag{10}$$

where $pcode$ is postfix code produced by function `translate`. The value of the application (10) is machine code.

Machine architecture comprises program counter PC, stack pointer SP (instead of stack pointer), accumulator A, the memory comprising target code $tcode$, and the stack $stack$.

The execution step is defined by the transformation of machine configuration (11).

$$
\begin{array}{l}
(PC,\ SP,\ A,\ tcode,\ stack) \Rightarrow \\
\quad \Rightarrow (PC',\ SP',\ A',\ tcode,\ stack')
\end{array}
\tag{11}
$$

Machine code execution is invoked by

$$\texttt{exec}\ (0, 0, 0, tcode, [])$$

where $tcode$ is target code loaded in memory.

We will not provide the Haskell functions, that implement the language, since they are trivial, we just attend, that the scheme $\mathcal{E}$ is implemented by function `pE`, the scheme $\mathcal{A}$ is implemented by function `pA`, and the scheme $\mathcal{B}$ is implemented by function `pB`.

However, below we introduce a simple example, which illustrates intermediate codes, the result of interpretation, as well as final configuration of machine architecture after execution, for source expression `2+3*5`, as obtained by Hugs98 interpreter. As defined by grammar (5), the multiplication is of higher priority than the addition, as expected.

```
> lexical "2+3*5"
[VL 2,AL,VL 3,ML,VL 5]
> translate [VL 2,AL,VL 3,ML,VL 5]
[Push 2,Push 3,Push 5,Mul,Add]
> eval [Push 2,Push 3,Push 5,Mul,Add]
17
> genload [Push 2,Push 3,Push 5,Mul,Add]
[5,2,5,3,5,5,3,1,0]
> exec (0,0,0,[5,2,5,3,5,5,3,1,0],[])
(8,1,17,[5,2,5,3,5,5,3,1,0],[17])
>
```

It may be noticed that `genload` adds **Exit** instruction (of code 0), to be able to stop the execution.

Since our implementation is functional, the interpreter of the language is the composition

$$\texttt{eval.translate.lexical}$$

and compiler, machine code generator, loader and execution is the application

$$\texttt{exec}(0, 0, 0, (\texttt{genload.translate.lexical})\texttt{s}, [])$$

where `s` is a source expression and composition

$$(\texttt{genload.translate.lexical})$$

translate source expression `s` to target code.

4 Adaptive Language

First, let us introduce the specific task of adaptation informally.

Depending on a result of interpretation (on the stack), the semantics (7) of language should be changed, so the next interpretation of the same source expression may yield different result.

The feedback loop depicted in Fig. 1 determines that we require new version for `translate` function, defined by (7), which is its zero variant (Variant 0).

Let the task of adaptation is as follows. If res is some result of interpretation, then we require next three variants:

1. If $res < 10$, then we require operations $(+)$ and $(-)$ to become right-associative, i.e., for variant 1, the rule $\mathcal{E}$ in (7) should change to the form (12).

$$\begin{aligned} &\mathcal{E}[\![\ A\ \underline{[}\ (\mathbf{AL} \mid \mathbf{SL})\ \underline{E}\ \underline{]}\ ]\!] = \\ &\quad \mathcal{A}[\![\ A\ ]\!]\ \underline{[}\ \underline{\mathcal{E}[\![\ E\ ]\!]}\ (\mathbf{Add} \mid \mathbf{Sub})\ \underline{]} \end{aligned} \tag{12}$$

2. If $10 \leq res < 20$ then we require $(*)$ and $(/)$ to become left-associative, for variant 1, the rule $\mathcal{A}$ in (7) should change to the form (13).

$$\begin{aligned} &\mathcal{A}[\![\ B\ \underline{\{}\ (\mathbf{ML} \mid \mathbf{DL})\ \underline{B'}\ \underline{\}}\ ]\!] = \\ &\quad \mathcal{B}[\![\ B\ ]\!]\ \underline{\{}\ \underline{\mathcal{B}[\![\ B'\ ]\!]}\ (\mathbf{Mul} \mid \mathbf{Div})\ \underline{\}} \end{aligned} \tag{13}$$

3. An finally, if $res \geq 20$, then mutual interchange of priority of $\{+,-\}$ and $\{*,/\}$ is required. It means that variant 3 requires changes of both rules $\mathcal{E}$ and $\mathcal{A}$, according (14).

$$\begin{aligned} &\mathcal{E}[\![\ A\ \underline{[}\ (\underline{\mathbf{ML}} \mid \underline{\mathbf{DL}})\ \underline{E}\ \underline{]}\ ]\!] = \\ &\quad \mathcal{A}[\![\ A\ ]\!]\ \underline{[}\ \underline{\mathcal{E}[\![\ E\ ]\!]}\ (\underline{\mathbf{Mul}} \mid \underline{\mathbf{Div}})\ \underline{]} \\ &\mathcal{A}[\![\ B\ \underline{\{}\ (\underline{\mathbf{AL}} \mid \underline{\mathbf{SL}})\ \underline{B'}\ \underline{\}}\ ]\!] = \\ &\quad \mathcal{B}[\![\ B\ ]\!]\ \underline{\{}\ \underline{\mathcal{B}[\![\ B'\ ]\!]}\ (\underline{\mathbf{Add}} \mid \underline{\mathbf{Sub}})\ \underline{\}} \end{aligned} \tag{14}$$

As follows from our requirement above, if a nonzero variant is sometimes selected, the language will never be adapted to its zero variant.

All changes above are underlined. Then we simply generalize rules for $\mathcal{E}$ and $\mathcal{A}$, implementing them by single function `gS`.

In addition, to this, adaptive function `translate` is sensitive on variant which is defined in metadata, by constant function `variant`.

Variants are defined in module `MetaData`, as shown in Fig. 4.

Adaptive translator is shown in Fig. 3.

The adaptiveness is reached by parameter

$$(\mathtt{s1}, \mathtt{t}, \mathtt{lo1}, \mathtt{o1}, \mathtt{lo2}, \mathtt{o2}, \mathtt{s2})$$

of `gS`, by function `rules`, and by function `ap`.

Function `rules` represents translation rules in a graph form, i.e. as a data and it is sensitive to variant. The translation rules are then applied indirectly – using function `ap`.

Adaptive translator is of type

$$\mathtt{translate} :: \mathtt{Int}- > \mathtt{LexSyms}- > \mathtt{PCode}$$

i.e. it is abstracted nonreflexive translator, of type

$$\mathtt{translate} :: \mathtt{LexSyms}- > \mathtt{PCode}$$

by variant number – a new parameter of type `Int`. Defining auxiliary function

$$translexk = (translatek).lexical$$

we may test the sensitivity to variants.

Expression $64/16/8 - 2 - 2$ is translated

- as $((64/(16/8)) - 2) - 2$ for variant 0,
- as $(64/(16/8)) - (2 - 2)$ for variant 1,
- as $(((64/16)/8) - 2) - 2$ for variant 2, and
- as $64/(16/((8 - 2) - 2))$ for variant 3, see below.

```
> translex 0 "64/16/8-2-2"
[Push 64,Push 16,Push 8,Div,Div,
Push 2,Sub,Push 2,Sub]
> translex 1 "64/16/8-2-2"
[Push 64,Push 16,Push 8,Div,Div,
Push 2,Push 2,Sub,Sub]
> translex 2 "64/16/8-2-2"
[Push 64,Push 16,Div,Push 8,Div,
Push 2,Sub,Push 2,Sub]
> translex 3 "64/16/8-2-2"
[Push 64,Push 16,Push 8,Push 2,Sub,
Push 2,Sub,Div,Div]
>
```

The implementation of adaptive language is shown in Fig. 2

Finally, we let us define metacode, in module `MetaCode`, see Fig. 5.

Selection criteria are defined by function `selVariant`, and as an example, we define function `adaptSeq`, which computes a list of pairs, first item being a variant number and second item being the result of evaluation, such

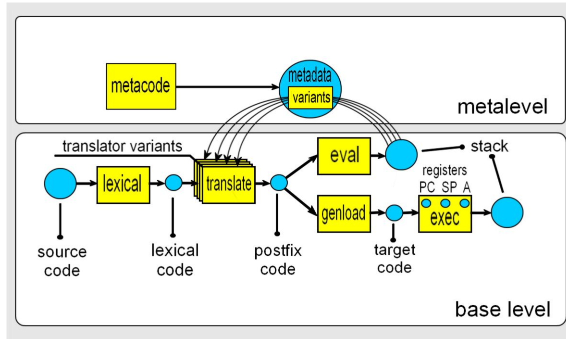

Figure 2: Adaptive Language

that corresponds to varying semantics dependent on previous result. We will compute original and four next results.

For example

```
> adaptSeq "64/16/8-2-2"
[(0,28),(3,16),(2,-4),(1,32),(3,16)]
```

so the value 28, computed by original (variant 0) semantics yields the selection of variant 3 (because $28 \geq 20$), and the same source expression will be recomputed in the second step with interchanged priority, i.e. as $64/(16/((8-2)-2))$, which yields the result 16. This result will determine variant 2 for the third step, hence we have the result (-4), etc.

The analysis of next two cases is left to a reader.

```
> adaptSeq "2+3*5"
[(0,17),(2,17),(2,17),(2,17),(2,17)]
> adaptSeq "2+3*8"
[(0,26),(3,40),(3,40),(3,40),(3,40)]
>
```

Substituting `evalS` by `loadS` or by `execS` we would see that adapted postfix code affects code generation, loading and the execution. It means that it is possible to decide first for interpretation and then produce highly efficient machine code. Of course, in our implementation is the architecture interpreted, so we have all resources accessible. In realistic computer machine there may be a problem with accessibility of internal registers, which is the main obstacle for building directly executable adaptive systems (and a reason for runtime environments).

As can be seen, our adaptive translator is more adaptable, than we have required. For example, it is possible to extend variants in `MetaData` to vary the semantics of lexical symbols (to obtain $*$ for addition and $+$ for multiplication).

5 Related Works

No matter how well an application is decomposed into modular entities, some functionality always crosscuts this modularization. This phenomenon is known as the tyranny of the dominant decomposition [7]. As a consequence, such functionalities cannot be evolved separately. The implementation convolution problem refers to the phenomenon that, for a large number of nontrivial functionalities, although their semantics are distinctive, their implementations do not have clear modular boundaries within the code space and, more seriously, often tangle with one another. This prohibits these functionalities from being pluggable [24].

Meta-modeling generic environments [17] provide the opportunity for multi-paradigm modeling, not however in combined manner. In spite of the principles of building software architectures were defined [20], there is no guarantee that the developed software system is without errors. Adding new aspects in the new lifecycle, some previous aspects may be forgotten, and removing indicated errors, new errors may be involved.

Aspect paradigm [1, 10, 2, 6, 16, 18, 22, 23] is based on modularization of aspects and their binding into implementation by weaving [9, 16]. The source code of aspect module is woven into different parts of target code during compile time in the case of static aspects or it is bound and executed in the run-time when some specific event occurs in the case of dynamic aspects. The set of join points is designated using logical conditions in pointcut designators. The main reason why the aspect paradigm is not widely accepted is the absence of general aspect model. Instead of one general methodology for development, monitoring and re-configuration of complex software systems, ad-hoc methods are applied nowadays. The programmer must take into account a lot of details of the system implementation, such as function names, parameters, types and even the order of execution - control flow to be competent to apply aspect oriented paradigm in everyday practice. These details have crosscutting nature because they represent new aspect as an added system property from the lexical, syntactic, semantic and run-time side of implementation.

Marginally, our work is related to the area of software evolution, such as dynamic software reconfiguration, on-line software evolution, or even dynamic unanticipated software evolution [19], but our approach differs in that our goal is not to change the software by program transformation, but transforming, i.e. adapting a language.

Current research is concentrating to the semantics of aspect languages. For example, the denotational seman-

tics of dynamic join points can be found in [22], a system for reasoning about temporal join points in [6], the analysis of AspectJ language [9] based on Scheme language in [16], and based on MiniMAO is introduced in [3]. Mining aspects and static refactoring are proposed in [7], as a basis for mutual interconnection of the specification and implementation. It has been shown that generic weaving based on repeated transformation of a program may fail in case of imperative assignments ad statement sequences [8]. Aspects in μABC calculus are primitive computational entities [2] and the state of computation is represented by a dynamically constructed advice.

A two-dimensional separation of concerns for compiler construction is introduced in [25] yields us to think about two dimensional domain specific language evolution. We feel the perspective of domain specific languages will grow as the domain can be understand not just as an entity associated to an application, but to the metalevels of an adaptive multiparadigm language. This direction seems to us better than semi-automatic approaches based on artificial intelligence evolution, such as in [5]. Each detail of a complex software system must be transparently reflected. Otherwise it is impossible to be responsible for its reliability. This however does not mean that reflection at a metalevel could not substitute the reflection at the base level.

6 Conclusion

Presented adaptive translator of trivial LL(1) language illustrates the ability for further extensions. First, it is possible to provide feedback loops from any subsequent phase of language implementation to any preceding phase, via metadata. For example, it is possible to replace names by feedback loop from translator, interpreter, machine code generator, loader, or even from target machine execution, since it is interpreted.

The weakness of presented solution is evident – its scattered definition of metalevel. But this is just for this reason, that we have used purely functional language without monads. A solution to this problem is functional language with monads, or an object oriented imperative language. Also low flexibility of algebraic types can be noticed, but this problem is over the scope of this paper.

On the other hand, it can be seen, that metalevel is formed by abstraction, which is implemented exactly as it is defined in terms of lambda calculus. Starting with original form of translator, i.e. with an expression e, we provide adaptive version by shifting original functions to local level (LETREC) of function representing new version of the translator. And this fact means exactly abstracting lamda expression e to lambda abstraction $(\lambda k.e)$, where k is a parameter designating any version.

In this paper, the generalization of two translation rules, is clearly ad-hock solution, and we plan extended it in the future for all syntactic trees, generated from extended BNF form (or BNF form for LR(k) languages). An alternative to denotational semantics – attributed grammars – can be also used.

The main contribution of this work, from the viewpoint of our future research, is as follows:

- Domain specific languages can be developed as adaptive language systems for rather metalevel domains than application domains.

- Provided that some level or metalevel is adaptive, it contains feedback loops from data to code via metalevel or metametalevel.

- Even if any level or metalevel is adaptive, it must be still manually initiated (i.e. programmed, specified, modeled). By the way, it is a base principle of control systems. The task of adaptive systems is to reduce this manual work, or shift it to the metalevel or metametalevel.

The prototype solution presented in this paper is just starting point to the research in emerging field of software language engineering. The idea is to treat the notion of change as a first-class entity in the language.

References

[1] Bebjak, M., Vranić V., Dolog, P. Evolution of Web Applications with Aspect-Oriented Design Patterns. Proc. of ICWE 2007 Workshops, 2nd International Workshop on Adaptation and Evolution in Web Systems Engineering, July 19, Como, Italy, pp. 80–86, 2007.

[2] Bruns, G., Jagadeesan, R., Jeffrey, A., Riely, J. μabc: A minimal aspect calculus. In Proceedings of the 2004 International Conference on Concurrency Theory, Springer–Verlag, pp. 209–224, 2004.

[3] Clifton, C., Leavens, G. T. MiniMAO: Investigating the semantics of proceed. FOAL 2005 Proceedings, Foundations of Aspect–Oriented Languages Workshop at AOSD 2005, pp. 51–61, 2005.

[4] Czarnecki, K., Eisenecker, U. E. Generative Programming: Methods, Tools, and Applications. Addison Wesley, 832 pp, 2005.

[5] Črepinšek, M., Mernik, M. Inferring Context-Free Grammars for Domain-Specific Languages, Conf. on Language Descriptions, Tools and Applications, LDTA 2005, April 3, Edinburgh, Scotland, UK, pp. 64–81, 2005.

[6] Douence, R., Motelet, O., Sudholt, M. A formal definition of crosscuts. In Reflection 2001, number 2192 in LNCS. Spring–Verlag, pp. 170–186, 2001.

[7] Ebraert, P., Tourwe, T. A Reflective Approach to Dynamic Software Evolution. In the proceedings of the Workshop on Reflection, AOP and Meta–Data for Software Evolution (RAM–SE'04), pp. 37–43, 2004.

[8] Fradet, P., Sudholt, M. Towards a Generic Framework for Aspect–Oriented Programming., Third AOP Workshop, ECOOP'98 Workshop Reader, LNCS, Vol. 1543, pp. 394–397, 1998.

[9] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W. G. An Overview of AspectJ. ECOOP'01, LNCS, vol. 2072, pp. 327–355, 2001.

[10] Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., Irwin, J. Aspect–oriented programming. In Mehmet Aksit and Satoshi Matsuoka, editors, ECOOP '97 – Object–Oriented Programming 11th European Conference, Jyväskylä, Finland, volume 1241, Springer–Verlag, , pp. 220–242, 1997.

[11] Kollár, J. Object Modelling using Process Functional Paradigm. Proc. ISM'2000, Rožnov pod Radhoštěm, Czech Republic, May 2–4, pp. 203–208, 2000.

[12] Kollár, J. Unified Approach to Environments in a Process Functional Programming Language. Computing and Informatics, 22, 5, pp. 439–456, 2003.

[13] Kollár, J., Porubän, J., Václavík, P. Separating Concerns in Programming: Data, Control and Actions. Computing and Informatics, 24, 5, pp. 441–462, 2005.

[14] Kollár, J., Novitzká, V. Semantical Equivalence of Process Functional and Imperative Programs. Acta Polytechnica Hungarica Vol. 1, No. 2, pp. 113–124, 2004.

[15] Kollár, J., Porubän, J., Václavík, P. From Eager PFL to Lazy Haskell, Computing and Informatics, 25, 1, pp. 61–80, 2006.

[16] Masuhara, H., Kiczales, G. Modeling crosscutting in aspect–oriented mechanisms. In ECOOP 2003 – Object–Oriented Programming European Conference, Springer–Verlag, pp. 2–28, 2003.

[17] Ledeczi, Á., Maroti, M., Bakay, A., Karsai, G., Garrett, J., Thomason, C., Nordstrom, G., Sprinkle, J., Volgyesi, P. The Generic Modeling Environment. Proc. of WISP'2001, May, Budapest, pp. 34–42, 2001.

[18] Lieberherr, K., Lorenz, D. H., Ovlinger, J. Aspectual Collaborations: Combining Modules and Aspects. The Computer Journal, 46(5), pp. 542–565, 2003.

[19] Oliveira de Almeida, H., Perkusich, A., Ferreira, G., Loureiro, E., de Barros Costa, E. A Component Model to Support Dynamic Unanticipated Software Evolution, SEKE 2006, San Francisco, CA, USA, pp. 262–267, 2006.

[20] Perry, D., Wolf, A. Foundations for the study of software architecture, ACM SIGSOFT Software Engineering Notes, 17, pp. 40–52, 1992.

[21] Steimann, F. The paradoxical success of aspect-oriented programming. OOPSLA 2006, pp. 481–497, 2006.

[22] Wand, M., Kiczales, G., Dutchyn, C. A semantics for advice and dynamic join points in aspect–oriented programming. Trans. on Prog. Lang. and Sys., 26(5), pp. 890–910, 2004.

[23] Walker, D., Zdancewic, S., Ligatti J. A theory of aspects. In Proceedings of the eighth ACM SIGPLAN international conference on Functional programming, Uppsala, Sweden, ACM Press, pp. 127–139, 2003.

[24] De Win, B., Piessens, F., Joosen, W., Verhanneman, T. On the importance of the separation–of–concerns principle in secure software engineering. Workshop on the Application of Engineering Principles to System Security Design, Boston, MA, USA, November 6–8, pp. 62–76, 2002.

[25] Xiaoqing Wu, Roychoudhury, S., Bryant, B. R., Gray, J. G., Mernik, M. A Two-Dimensional Separation of Concerns for Compiler Construction. Proceedings of the 2005 ACM symposium on Applied computing, pp. 1365–1369, 2005.

7 Appendix

```
module Translator where

import Lexical
import MetaData

translate :: Int -> LexSyms -> PCode
translate k ls = (snd . pE) (ls,[])
 where
  rules = [("E", gS v1), ("A", gS v2), ("B", pB) ]
          where (v1,v2) = variants !! k

  ap nt = snd (head [(n,f) | (n,f) <- rules , n==nt])

  pE = ap "E"

  gS :: (String,Char,LexSym,Operation,LexSym,Operation,String)->
        (LexSyms,PCode) -> (LexSyms,PCode)
  gS (s1,t,lo1,o1,lo2,o2,s2) ([],cs)  = ([],cs)
  gS (s1,t,lo1,o1,lo2,o2,s2) (ls,cs)
    | t == 'c' = cls ((ap s1 ) (ls,cs)) []
    | t == 'a' = alt ((ap s1 ) (ls,cs)) []
     where cls ([], cs) no = ([], cs++no)
           cls ((l:ls), cs) no
                | l == lo1 = cls ((ap s2 ) (ls,cs++no) ) [o1]
                | l == lo2 = cls ((ap s2 ) (ls,cs++no) ) [o2]
                | otherwise = ((l:ls),cs++no)
           alt ([], cs) os  = ([], cs++os)
           alt ((l:ls), cs) os
                 | l == lo1 = alt ((ap s2 ) (ls,cs)) (os++[o1])
                 | l == lo2 = alt ((ap s2 ) (ls,cs)) (os++[o2])
                 | otherwise = ((l:ls),cs++os)

  pB :: (LexSyms,PCode) -> (LexSyms,PCode)
  pB ([],cs)                = ([],cs)
  pB (((VL x):ls),cs)       = (ls,cs++[Push x])
  pB ((l:ls),cs) | l == LL = skipR ((ap "E" ) (ls,cs))
                             where
                               skipR ((l:ls'),cs') = (ls',cs')
```

Figure 3: Adaptive translator to postfix code

```
module MetaData where

import Lexical -- to access LexSyms type

data Operation = Add | Sub | Mul | Div | Push Int

type PCode = [Operation]

variants = [(r1,r2), (r3,r2), (r1,r4), (r5,r6)]
             where
               r1 = ("A",'c',AL,Add,SL,Sub,"A")
               r2 = ("B",'a',ML,Mul,DL,Div,"A")
               r3 = ("A",'a',AL,Add,SL,Sub,"E")
               r4 = ("B",'c',ML,Mul,DL,Div,"B")
               r5 = ("A",'a',ML,Mul,DL,Div,"E")
               r6 = ("B",'c',AL,Add,SL,Sub,"B")
```

Figure 4: `MetaData` module – definition of variants

```
module MetaCode where

import Lexical
import Translator
import Evaluation

selVariant v | v <  10           = 1
             | v >= 10 && v < 20 = 2
             | v >= 20           = 3

evalS k = eval . (translate k) . lexical

loadS k = genload . (translate k) . lexical

execS k s = exec (0,0,0,(genload . (translate k) . lexical) s,[])

variate s (vs,res) ls  0    = ls
variate s (vs,res) ls (n+1) = variate s (vs',res') (ls ++ [(vs',res')]) n
  where
    res' = evalS vs' s
    vs'  = (selVariant res)

adaptSeq s = variate s (vs,res) [(vs,res)] 4
  where
    res = evalS 0 s
    vs  = 0
```

Figure 5: `MetaCode` module – user defined metacode