UDC 004.43

Functional Approach to the Adaptation of Languages instead of Software Systems*

Ján Kollár[1], Jaroslav Porubän[1], Peter Václavík[1], Jana Bandáková[1], and Michal Forgáč[1]

[1] Department of Computers and Informatics, Technical University, Letná 9, 04200 Košice, Slovakia
{Jan.Kollar, Jaroslav.Poruban, Peter Vaclavik, Jana.Bandakova, Michal.Forgac}@tuke.sk

Abstract. 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. Multiple metalevel concepts are 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 sense, practical experiment introduced in this paper is related to the base level of language, but it illustrates the ability for extensions primarily in horizontal but also in vertical direction of an adaptive system.

1. Introduction

There is an increasing demand for systems that can be easily configured for a specific environment or they even adjust themselves dynamically to a changing environment at runtime. Adaptive behaviour is the proposition for the runtime adjustment, or evolution, provided that it is performed automatically.

Adaptability [8] is mostly related to the area of software engineering – object oriented programming, aspect-oriented programming, intentional programming, template programming, etc., where it is exploited for changing the semantics of programs.

On the other hand, the properties of systems are expressed using various languages, such as programming languages, specification languages, or modelling languages, in that solutions of problems are constructively formulated, no matter of a language abstraction level. At the bottom level of a

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

system, the machine code written in a machine language is executed. At the top level, human thoughts arise and are formulated using a natural language.

Our current research concentrates on how a language (not a program) can vary its semantics, reflecting not just compile time but also runtime events. According to our opinion, static and dynamic adaptation of the language to new software aspects and runtime events should exclude current expensive methods of compiler construction. The language should be minimal, strongly associated with the properties of a software system and should be adaptive.

This is a step on a way to self-adaptive software – software that incorporates monitoring and evaluation functions, and can rapidly (at runtime) respond to some sorts of need for change [9].

In a simplified manner, the ascending goal of our research is described below.

Suppose *mExec* is a hardwired (non-adaptable) computer architecture, which executes the target code *trgCode* obtained from the source code *srcCode* using the language processor *langProc*.

If the semantics of *trgCode* is not equivalent to the semantics of *trgCode'*, for two different target codes ($[\![trgCode]\!] \not\equiv [\![trgCode']\!]$), then these target codes yield different behaviour when executed, accordingly (1).

$$[\![mExec\ trgCode]\!] \not\equiv [\![mExec\ trgCode']\!] \qquad (1)$$

Clearly, *langProc* implements a language *L*. The basic principle of language processing is such that semantics of a source code is the same as the semantics of the target code produced by *langProc* from this source code. This is expressed by equivalence (2).

$$[\![srcCode]\!] \equiv [\![trgCode]\!] \qquad (2)$$

The semantic equivalence (2) follows directly from the equation (3), which defines the target code in terms of the application of a language processor to the source code.

$$trgCode = langProc\ srcCode \qquad (3)$$

Then, instead of a manual development of a software system, represented by the step (4), changing the source program,

$$mExec\ (\ langProc\ srcCode\ ) \Rightarrow mExec\ (\ langProc\ srcCode'\ ) \qquad (4)$$

we would like to change the language processor, preserving the original source program, replacing the manual development step (4) by a semi-automatic evolution step (5)

$$mExec\ (\ langProc\ srcCode\ ) \Rightarrow mExec\ (\ langProc'\ srcCode\ ) \qquad (5)$$

such that the target behaviour is the same, i.e. the equivalence (1) holds.

$$[\![mExec\ (\ langProc\ srcCode'\ )]\!] \equiv [\![mExec\ (\ langProc'\ srcCode\ )]\!] \qquad (6)$$

The benefit is clear. Then we would be able to develop the systems without changing the source code. The trouble is that we must still add some additional program or specification to change the language (implemented by language processor), so the evolution is not fully automatic, but rather semi-automatic. On the other hand, our hypothesis, which we would like to prove and subsequently exploit, is that manual code increment which adapts the language is far smaller, and the style of adaptation is far more systematic, as when source programs are manually modified. It may be also noticed, that (6) expresses the equivalence of runtime behavior, so we are interested not just in some slow process of adaptation, but even in a very fast adaptation of languages in runtime.

Thinking about the task above, we have recognized the concepts of *metaprogramming* and *reflection* are fundamental. *Metaprogramming* is about writing programs that represent and manipulate other programs or themselves, i.e. metaprograms are programs about programs [2]. *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 primary subject matter [4]. 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.

In this paper, we present the principle of an adaptive context-free language focusing on the most complex phase – the syntax-directed translation from lexical symbols to postfix code. For the purpose of simplicity and clarity, we decided to use Haskell without monads.

In Section 2 we introduce our classification of software systems from the viewpoint of the degree of reflexive behaviour, and we analyze three selected cases. In Section 3 we present LL(1) language for expressions and its nonreflexive implementation. The main ideas for adaptive implementation of the system are presented in Section 4. The conception of the multilevel adaptive language system is discussed briefly in Section 5.

2. Systems Behaviour Classification

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

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 ${}^0\mathbf{D}^k$ (a 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 (7).

$$ {}^0\mathbf{C} \xrightarrow{\times} {}^0\mathbf{D}^k \Rightarrow {}^0\mathbf{C} \xrightarrow{\times} {}^0\mathbf{D}^{(k+1)} \qquad (7) $$

In (7), the relation ($\xrightarrow{\times}$) denotes that multiple instructions from ${}^0\mathbf{C}$ can access multiple data records ${}^0\mathbf{D}$.

The execution is nonreflexive, if there is no feedback loop from data to code, and no possibility is 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 a nonreflexive execution. In addition to this, each subset of the set ${}^0\mathbf{D}^k$ refers to (which we designate by $\xrightarrow{-}$) exactly one element of static data ${}^1\mathbf{S}$, and this data refers ($\xrightarrow{+}$) to a subset of code ${}^0\mathbf{C}$ accordingly (8). Static data set ${}^1\mathbf{S}$ at level 1 is metalevel static data to the level 0.

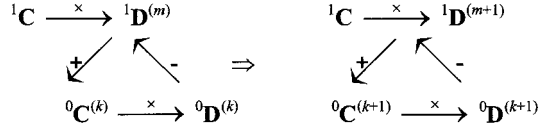
$$ \begin{array}{ccc} {}^1\mathbf{C} & \xrightarrow{\times} & {}^1\mathbf{S} \\ & {\scriptstyle +}\swarrow \quad \nwarrow{\scriptstyle -} & \\ {}^0\mathbf{C} & \xrightarrow{\times} & {}^0\mathbf{D}^k \end{array} \quad \Rightarrow \quad \begin{array}{ccc} {}^1\mathbf{C} & \xrightarrow{\times} & {}^1\mathbf{S} \\ & {\scriptstyle +}\swarrow \quad \nwarrow{\scriptstyle -} & \\ {}^0\mathbf{C} & \xrightarrow{\times} & {}^0\mathbf{D}^{(k+1)} \end{array} \qquad (8) $$

Since metalevel data (set of records) ${}^1\mathbf{S}$ is static, metacode ${}^1\mathbf{C}$ may produce it just 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 the existence of a feedback loop from the code ${}^0\mathbf{C}$ to code ${}^0\mathbf{C}$ via the data set ${}^0\mathbf{D}$ and some metadata element from ${}^1\mathbf{S}$. For example, introspective (but not adaptive) behaviour can be obtained using Java's metaclasses that exploit static metadata ${}^1\mathbf{S}$.

Introspection enables monitoring but not the behavioural change, so this property is of less significance for us. Moreover, each adaptive system, which execution is described below is automatically introspective.

2.3. Adaptive Execution

Adaptive execution step is defined in (9). 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 metametalevel 2. On the other hand, a new ${}^1\mathbf{D}^{(m+1)}$ may yield a new ${}^0\mathbf{C}^{(k+1)}$, continuing its execution at level 0.



(9)

In this way, code at level 0 may be not just introspective, but also adaptive, and this fact is essential for an adaptive execution. For example, adaptive behaviour can be obtained using Smalltalk's metaobjects that exploit dynamic metadata ${}^1\mathbf{D}^{(m)}$.

3. Nonreflexive Language Implementation

In this section, we introduce the implementation of simple LL(1) language for expressions in a nonreflexive manner.

Grammar of the language is written in extended BNF (EBNF), see (10).

$$\begin{aligned} E &\rightarrow A\ \{\ (\text{"+"} \mid \text{"-"})\ A\ \} \\ A &\rightarrow B\ [\ (\text{"*"} \mid \text{"/"})\ A\ ] \\ B &\rightarrow const \mid \text{"("}\ E\ \text{")"} \end{aligned} \qquad \textbf{(10)}$$

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

Lexical analyzer and translator to postfix code are common passes to both compiler and interpreter of the language.

Lexical analyzer translates lexical units from string to symbol form, for example "+" to **AddL**, "-" to **SubL**, integer constants *const* to **ValL** *v*, etc. The correspondence of string and symbol form for all lexical units is visible by comparison of (10) and (11).

The translator `translate` translates symbol form of lexical units to postfix code using syntax directed translation, following the rules (11).

$$
\begin{array}{l}
\boldsymbol{E}[\![A\ \{\ \textbf{AddL}\ A_k\ \}\ ]\!] = \boldsymbol{A}[\![A\ ]\!]\ \ \{\ \boldsymbol{A}[\![A_k\ ]\!]\ \textbf{Add}\ \} \\
\boldsymbol{E}[\![A\ \{\ \textbf{SubL}\ A_k\ \}\ ]\!] = \boldsymbol{A}[\![A\ ]\!]\ \ \{\ \boldsymbol{A}[\![A_k\ ]\!]\ \textbf{Sub}\ \} \\
\boldsymbol{A}[\![B\ [\ \textbf{MulL}\ A\ ]\ ]\!] = \boldsymbol{B}[\![B\ ]\!]\ \ [\ \boldsymbol{A}[\![A\ ]\!]\ \textbf{Mul}\ ] \\
\boldsymbol{A}[\![B\ [\ \textbf{DivL}\ A\ ]\ ]\!] \quad = \boldsymbol{B}[\![B\ ]\!]\ \ [\ \boldsymbol{A}[\![A\ ]\!]\ \textbf{Div}\ ] \\
\boldsymbol{B}[\![\ \textbf{ValL}\ v\ ]\!] \quad\quad = \textbf{Push}\ v \\
\boldsymbol{B}[\![\ \textbf{LparL}\ E\ \textbf{RparL}\ ]\!] = \boldsymbol{E}[\![E\ ]\!]
\end{array}
\qquad (11)
$$

The translation starts with $\boldsymbol{E}[\![E\ ]\!]$, since E is starting symbol.

Compiler consists of lexical analyser, translator, machine code generator and loader. The execution is performed by emulated target machine.

Code generation and loading are composed into single pass. The transformation of postfix code to machine code by $\boldsymbol{C}$ is defined in (12).

$$
\begin{array}{lll}
\boldsymbol{C}[\![\textbf{Add}\ ]\!] = 1 & \boldsymbol{C}[\![\textbf{Sub}\ ]\!] = 2 & \boldsymbol{C}[\![\textbf{Mul}\ ]\!] = 3 \\
\boldsymbol{C}[\![\textbf{Div}\ ]\!] = 4 & \boldsymbol{C}[\![\textbf{Push}\ x\ ]\!] = 5\ x &
\end{array}
\qquad (12)
$$

where x is integer value, pushed on the stack by instruction **Push** (code 5). Machine code is generated and loaded to memory by function `genload`. In this way, `genload` performs code generation as well as loading actions that are invoked by the application (13)

$$\texttt{genload}\ \textit{pcode} \qquad (13)$$

where *pcode* is postfix code produced by translator `translate`. The value of the application (13) is machine code.

Exit instruction is added (i.e. woven) to machine code in load time to enable to stop the execution. In this way, the semantics of original LL(1) language is statically (although not significantly) changed.

Machine architecture comprises program counter *pc*, the number of stacked values *sp* (used instead of stack pointer), the accumulator *a*, the memory *mem*, and the stack *stack*. An execution step is defined by the transformation of machine configuration (14).

$$(pc,\ sp,\ a,\ mem,\ stack) \Rightarrow (pc',\ sp',\ a',\ mem',\ stack') \qquad (14)$$

Machine code is executed (emulated) by `exec (0,0,0,mcode,[])`, where `mcode` is target code loaded in memory *mem*.

```
translate :: [LexUnit] -> [Instruction]
translate ls = (snd . pE) (ls,[])
```

```
pE :: (LexUnits,Code) -> (LexUnits,Code)
pE ([],cs) = ([],cs)
pE (ls,cs) = cls (pA (ls,cs)) []
  where cls ([], cs) no = ([], cs++no)
        cls ((l:ls), cs) no
          | l == AddL = cls (pA (ls,cs++no) ) [Add]
          | l == SubL = cls (pA (ls,cs++no) ) [Sub]
          | otherwise = ((l:ls),cs++no)

pA :: (LexUnits,Code) -> (LexUnits,Code)
pA ([],cs) = ([],cs)
pA (ls,cs) = alt (pB (ls,cs)) []
  where alt ([], cs) os = ([], cs++os)
        alt ((l:ls), cs) os
          | l == MulL = alt (pA (ls,cs)) (os++[Mul])
          | l == DivL = alt (pA (ls,cs)) (os++[Div])
          | otherwise = ((l:ls),cs++os)

pB :: (LexUnits,Code) -> (LexUnits,Code)
  pB ([],cs)                    = ([],cs)
pB (((ValL x):ls),cs)     = (ls,cs++[Push x])
pB ((l:ls),cs) | l == LparL = skipR (pE (ls,cs))
  where skipR ((l:ls'),cs') = (ls',cs')
```

Fig. 1. Nonreflexive translator to postfix code

Interpreter consists of lexical analyser, translator, and function `eval`, which evaluates postfix code *pcode* directly, according to (15).

$$\texttt{eval}\ pcode = v \qquad \textbf{(15)}$$

in which *pcode* is postfix code produced by the translator and v is the result of interpretation.

Although compiler and machine emulator have great potential for adaptive implementation, we will focus on adaptive language using interpreter, for the limited scope of this paper. A nonreflexive version of interpreter `interpreter` is defined by composition, as follows:

```
interpreter = eval . translate . lexical        (16)
```

Table 1. The task of adaptation

Variant	On condition	Requirement
0	none	none
1	*res* < 10	$\{+, -\} \rightarrow R$
2	*res* $\in \langle 10, 20)$	$\{*, /\} \rightarrow L$
3	*res* ≥ 20	$\{+, -\} \leftrightarrow \{*, /\}$

Since in the next section we will concentrate to the adaptation of function translate, which will be generalized, we introduce its nonreflexive version in Fig. 1. In this version, functions `pE`, `pA`, and `pB` implement the translation schemes $\boldsymbol{E}$, $\boldsymbol{A}$, and $\boldsymbol{B}$, see (11).

4. Adaptive Language Implementation

First, let us introduce the task of adaptation informally.

> *Depending on the result of interpretation, the language defined by* (10) *and* (11) *should be changed, and the next interpretation follows different semantics, i.e. potentially different result of the same source expression.*

We have selected this task taking into account that it affects the most complex phase – the translation of context-free language to postfix language.

More specifically, let *res* be a result of interpretation. If $res < 10$, then we require operations (+) and (-) be right-associative. If $res \in \langle 10, 20)$ then we require (*) and (/) be left-associative. And finally, if $res \geq 20$, then mutual interchange of priority of {+, -} and {*, /} is required.

All mentioned requirements are summarized in Table 1, which contains also zero variant, corresponding to original priority and associativity of operations defined by (10), as follows: operations (+) and (-) are left-associative, and they are of lower priority than operations (*) and (/), that are right-associative.

According to our specification, if a non-zero variant is selected, the language will never be adapted to its zero variant.

First, we generalize the translator, using the following methodology: Comparing the translation schemes of rules for E and A, we define more general implementation for two rules in the form of function `gS`, introduced in Fig. 2.

The adaptability is reached by parameter `(s1,t,lo1,o1,lo2,o2,s2)` of `gS`, by function rules, and by function `ap`.

Looking at (10) and (11), the meaning of parameter `(s1,t,lo1,o1,lo2,o2,s2)` items is as follows:

`s1` is the first nonterminal, which represents the first occurrence of A in E rule and B in A rule,

`t` . . . closure { } or alternative [ ] switch,

`lo1` . . . input coding for the first operator, which represents **AddL** and **MulL**,

`o1` . . . output coding for the first operator, which represents **Add** and **Mul**,

`lo2` . . . input coding for the second operator, which represents **SubL** and **DivL**,

`o2` . . . output coding for the second operator, which represents **Sub** and **Div**,

`s1` . . . the second nonterminal, the second occurrence of A in E rule, and B in A rule.

Function `rules` represents translation rules in a graph form, i.e. as a data and it is sensitive to the variant. The translation rules are then applied indirectly – using function `ap`. Variants that affect the translation rules, performing translator adaptation, are shown in Fig. 3.

Finally, we define a simple metacode, including also adaptive interpreter `interpreter`, see Fig. 4. Using auxiliary function `allVariants`, we can verify correctness of adaptive interpretation for all variants, for example:

```
> allVariants "10-3-1"
= [6,8,6,6]
> allVariants "20-3-1"
= [16,18,16,16]
> allVariants "30-3-1"
= [26,28,26,26]
```

Function `adaptInt` performs the change of the language semantics according to the selected variant `k`, dependent on previous result of evaluation.

Function `adaptInt` takes a source expression and produces the triple: the first item is the value of the source expression, the second item is selected variant number depending on this value. The third item – new value is obtained by interpretation of the same source expression translated to potentially different postfix code by adapted translator and subsequently evaluated.

For example, for input expression (10 - 3 - 1) (of value 6), variant 1 is selected and expression will be re-evaluated as being in the form (10 - (3 - 1)), producing value 8.

This is so, because variant 1 selects pair `(v3,v2)` from variants, see Fig. 3, and adapts scheme ε in (11) to the scheme

$$E[\![A\ [\ \mathbf{AddL}\ E\ ]\ ]\!] = A[\![A\ ]\!]\ \ [\ E[\![E\ ]\!]\ \mathbf{Add}\ ]$$

$$E[\![A\ [\ \mathbf{SubL}\ E\ ]\ ]\!] = A[\![A\ ]\!]\ \ [\ E[\![E\ ]\!]\ \mathbf{Sub}\ ]$$

For input expression (14 - 3 - 1) (of value 10), variant 2 is selected and the expression will be re-evaluated as ((14 - 3) - 1), producing the same value 10. Although variant 2 adapts scheme A to the scheme

$$A[\![B\ \{\ \mathbf{MulL}\ B_k\}\ ]\!] = B[\![B\ ]\!]\ \ \{\ B[\![B_k\ ]\!]\ \mathbf{Mul}\ \}$$
$$A[\![B\ \{\ \mathbf{DivL}\ B_k\}\ ]\!] = B[\![B\ ]\!]\ \ \{\ B[\![B_k\ ]\!]\ \mathbf{Div}\ \}$$

this semantical change does not affect the expression, in which just subtraction is applied.

The evaluation in both cases described above is as follows:

```
> adaptInt "10-3-1"
= (6,1,8)
> adaptInt "14-3-1"
= (10,2,10)
```

But notice, `adaptInt "64/8/2"` would evaluate first time to 16, but for the second time to 4.

Our metacode can be extended for solving more powerful tasks, and different metadata variants may result to semantically different adaptation effects. It can be also noticed, that adaptive language implementation is stronger than an introspective one. For example, an introspective task is such as counting the number of addition operations used in an expression.

5. Discussion and Related Works

The work presented in this paper comes out from our past research in the application of our process functional paradigm [6,7] to the aspect-oriented languages [5,10,12], until we have recognized that statically defined semantics of the language of pointcut designators weakly supports the adaptability, which we follow. The detailed analysis of this fact is over the scope of this paper, but having performed this we have decided to return back to Lieberherr's [8] concept of adaptive systems, and even to their essential principles, such as metaprogramming and reflection. Some ideas about application of aspect-oriented programming to software evolutionary changes can be found in [1].

In this paper, we have used a purely functional approach using Haskell [11], but without monads [13], to simplify the notation as much as possible. Close relation of our adaptive language implementation and adaptive execution (9) is still visible: we abstract multiple translators by single translate

k function, but this is equivalent to the association of a new version of translator via a new metadata.

A two-dimensional separation of concerns for compiler construction [14] tends us to think about multi-dimensional domain specific language evolved in multiple metalevels.

```
translate :: Int -> [LexUnit] -> [Instruction]
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, LexUnit,Instruction,LexUnit,
Instruction,String) -> (LexUnits,Code) -> (LexUnits,Code)
  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 :: (LexUnits,Code) -> (LexUnits,Code)
  pB ([],cs)              = ([],cs)
  pB (((ValL x):ls),cs)     = (ls,cs++[Push x])
  pB ((l:ls),cs)
   | l == LparL = skipR ((ap "E" ) (ls,cs))
    where
     skipR ((l:ls'),cs') = (ls',cs')
```

Fig. 2. Adaptive translator to postfix code

```
variants = [(v1,v2), (v3,v2), (v1,v4), (v5,v6)]
 where v1 = ("A",'c',AddL,Add,SubL,Sub,"A")
```

```
v2 = ("B",'a',MulL,Mul,DivL,Div,"A")
v3 = ("A",'a',AddL,Add,SubL,Sub,"E")
v4 = ("B",'c',MulL,Mul,DivL,Div,"B")
v5 = ("A",'a',MulL,Mul,DivL,Div,"E")
v6 = ("B",'c',AddL,Add,SubL,Sub,"B")
```

Fig. 3. Definition of variants

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

interpreter k = eval . translate k . lexical

adaptInt s = (res, variant, interpreter variant s)
           where
             res     = interpreter 0 s
             variant  = (selVariant res)

allVariants s = [interpreter 0 s,
                 interpreter 1 s,
                 interpreter 2 s,
                 interpreter 3 s]
```

Fig. 4. Metalevel code

This approach has sense, if each higher metalevel generalizes lower metalevel very concisely, and the computational time does not increase significantly.

In contrast to language evolution by inferring a language from samples of programs [3], our approach is based on inferring a metalanguage from samples of metaprograms.

6. Conclusion

We have presented the classification of software systems considering the degree of adaptibility. We recognize non-reflexive, introspective and adaptive systems. The most powerful case of behaviour – adaptive behaviour is analyzed and implemented using a simple LL(1) language. Non-reflexive interperter of this language, written in Haskell functional language, is transformed, and its adaptive version is obtained. In this way we provide a language able to react to the run-time event.

Presented adaptive LL(1) language can be extended in many directions, exploiting feedback loops from any phase to any phase of compiler, via metadata. Using an object-oriented language, adaptive behaviour (9) would be directly implemented, instead of current abstraction of an expression e to $(\lambda k.e)$, where k is a parameter designating a version. Presented generalization of two translation rules is just ad-hoc solution, and it is necessary to extend it to all constructs of EBNF or BNF. Instead of nonreflexive interpreter

```
interpreter = eval . translate . lexical
```

using abstraction and generalization, we have developed adaptive interpreter, as follows.

```
interpreter k = eval . translate k . lexical
```

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

Provided that a level or metalevel is adaptive, it contains feedback loops from data to code via metalevel or metametalevel, respectively. Even if any level or metalevel is adaptive, it still must be manually initiated (programmed, specified, modelled). By the way, this is an essential principle of control systems. The task of adaptability is to reduce this manual work, or to shift it to the higher metalevels. There is no need for a universal language, just for a multi-metalevel domain specific language, which is able to express current and future properties of a system accurately.

We may conclude, that the most significant, except the generalization of our ad-hoc use of extended BNF form and denotational semantics, is the extension to any metalevel l for ${}^{l}\mathbf{C}$, ${}^{l}\mathbf{D}$ and combining of ${}^{l}\mathbf{C}^{P}$, ${}^{l}\mathbf{D}^{P}$ at the same level, considering different programming, specification, and modeling paradigms $\boldsymbol{P}$.

This however is impossible to do exploiting purely functional approach, which we have used in this paper to illustrate how a simple language can be adaptive. Instead of that, at least two other approaches come into account, using monadic functional or metaobjects languages.

7. References

1. Bebjak, M., Vranić, V., Dolog, P.: Evolution of Web Applications with Aspect-Oriented Design Patterns. Proc. of the 2nd International Workshop on Adaptation and Evolution in Web Systems Engineering (AEWSE'07), Como, Italy, July 2007. CEUR Workshop Proceedings, ISSN 1613-0073. http://CEUR-WS.org/Vol-267/paper7.pdf.
2. Czarnecki, K., Eisenecker, U.E.: Generative Programming: Methods, Tools, and Applications. Addison Wesley (2000), 832 pp.

3. Črepinšek, M., Mernik, M.: Inferring Context-Free Grammars for Domain-Specific Languages, Conf. on Language Descriptions, Tools and Applications, LDTA 2005, April 3, 2005, Edinburgh, Scotland, UK, pp. 64–81.
4. 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), June 2004, pp. 37–43.
5. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An Overview of AspectJ. ECOOP'01, 2001, LNCS, vol. 2072, pp. 327–355.
6. Kollár, J.: Object Modelling using Process Functional Paradigm. Proc. ISM'2000, Rožnov pod Radhoštěm, Czech Republic, May 2–4, 2000, pp. 203–208.
7. Kollár, J.: Unified Approach to Environments in a Process Functional Programming Language. Computing and Informatics, 22, 5, 2003, pp. 439–456.
8. Lieberherr, K.: Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns (1996), Northeastern University May 23, 1995, 33pp.
9. Laddaga, R., Robertson, P., Shrobe, H.: Self-Adaptive Software: Internalized Feedback. Chapter 26 in Software Evolution and Feedback: Theory and Practice, Wiley, 2006, 612 pp. ISBN 0470871806.
10. Masuhara, H., Kiczales, G.: Modeling crosscutting in aspect–oriented mechanisms. In ECOOP 2003 – Object–Oriented Programming European Conference,Springer–Verlag, 2003, pp. 2–28.
11. Peyton Jones, S.L.,Hughes, J. [editors]: Report on the Programming Language Haskell 98 - A Non-strict, Purely Functional Language. February 1999, 163 pp.
12. Steimann, F.: The paradoxical success of aspect-oriented programming. OOPSLA 2006, 2006, pp. 481–497.
13. 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.
14. Wu, W., 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, 2005, pp. 1365–1369.

Ján Kollár is Associate Professor of Informatics at Department of Computers and Informatics, Technical university of Košice, Slovakia. He received his M.Sc. summa cum laude in 1978 and his Ph.D. in Computer Science in 1991. In 1978-1981 he was with the Institute of Electrical Machines in Košice. In 1982-1991 he was with Institute of Computer Science at the P.J. Šafárik University in Košice. Since 1992 he is with the Department of Computer and Informatics at the Technical University of Košice. In 1985 he spent 3 months in the Joint Institute of Nuclear Research in Dubna, USSR. In 1990 he spent 2 months at the Department of Computer Science at Reading University, UK. He was involved in research projects dealing with real-time systems, the design of microprogramming languages, image processing and remote sensing, dataflow systems, implementation of programming languages, and high performance computing. He is the author of process functional programming paradigm. Currently his research area covers formal languages and automata, programming paradigms, implementation of programming languages, functional programming, and adaptive software and language evolution.

Jaroslav Porubän is Assistant Professor of Informatics at Department of Computers and Informatics, Technical university of Košice, Slovakia. He received his MSc. in 2000 and his Ph.D. in Computer Science, in 2004. Since 2003 he is with the Department of Computers and Informatics at Technical University of Košice. He was involved in the research of profiling based on process functional language. The subjects of his research are formal languages and automata, implementation of programming languages, programming and modeling paradigms, functional and parallel programming, aspect-oriented languages, generic programming and modeling, and adaptive software and language evolution.

Peter Václavík is Assistant Professor of Informatics at Department of Computers and Informatics, Technical university of Košice, Slovakia. He received his MSc. in 2000 and his Ph.D. in Computer Science, in 2004. Since 2003 he is with the Department of Computers and Informatics at Technical University of Košice. He was implementing object oriented version of PFL language. The subjects of his research are programming paradigms, abstract types, functional programming, aspect-oriented languages, aspect programming, and adaptive software and language evolution.

Jana Bandáková is doctoral student at Department of Computers and Informatics, TU of Košice, Slovakia. She received her MSc. in Informatics at Technical University of Košice, Slovakia, in 2005. The subjects of her research are modeling paradigms and the development of adaptive models in software evolution.

Michal Forgáč is doctoral student at Department of Computers and Informatics, TU of Košice, Slovakia. He received his MSc. in Informatics at Technical University of Košice, Slovakia, in 2006. The subject of his research is metaprogramming, programming paradigms, and systems evolution by run-time adaptation.