ABSTRACTION IN PROGRAMMING LANGUAGES ACCORDING TO DOMAIN-SPECIFIC PATTERNS

Ján KOLLÁR, Emília PIETRIKOVÁ, Sergej CHODAREV
Department of Computers and Informatics, Faculty of Electrical Engineering and Informatics, Technical University of Košice,
Letná 9, 042 00 Košice, Slovak Republic, tel.: +421 55 602 2577,
e-mail: jan.kollar@tuke.sk, emilia.pietrikova@tuke.sk, sergej.chodarev@tuke.sk

ABSTRACT

This paper focuses on the presentation of an approach to language pattern recognition and distinguishes two concepts: abstraction and structural complexity. Abstraction as well as language patterns are examined from both current state and future perspectives. The abstraction is traced through a specific set of tools, and the premises on which the measuring methodology stands are critically analyzed with respect to theoretical and application concerns. Particular attention is paid to the main features that characterize language patterns, proposing a method for automatized raise of abstraction level based on recognition of patterns in program source code (thus not design patterns), with contribution to a new approach in development of programming languages. In addition, a large group of programs is examined with the goal of predicting the future development and application of the language patterns. All the presented experiments are performed within a specific domain of programs, providing sample derivation trees.

Keywords: *Abstraction, list comprehension, complexity, syntactic patterns, domain-specific languages*

1. INTRODUCTION

Programmers need to deal with a great amount of complexity [1]. With growth of software systems, expression complexity of their properties in a programming language mounts up as well. As the answer to complexity, higher levels of abstraction can be introduced. Abstraction allows expressing problems more simply by defining new, more abstract concepts that encapsulate complex expressions. This allows to hide implementation details. Therefore, a promising solution for growth of program complexity can be an abstraction based on a language, allowing reduction of the complexity through definition of new, more abstract concepts and language constructions.

This way of problem solution is divided into several levels, where each level provides abstractions for the above level, also called "stratified design" [2]. Provided that lower levels are already in place, we can concentrate on problem solution. The role of abstraction and structural complexity within hierarchy of hardware and software systems is depicted in Fig. 1.

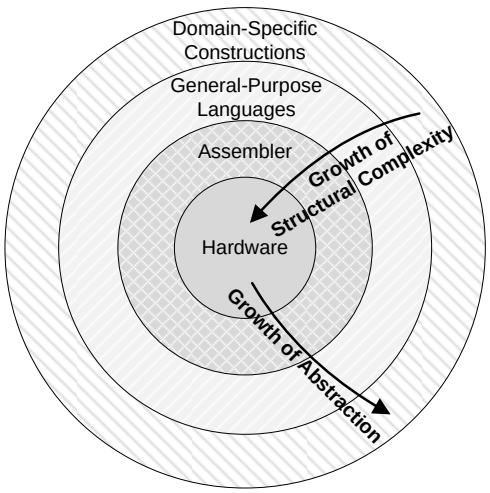

Fig. 1 Abstraction and structural complexity within hierarchy of software and hardware systems

Programming languages are also part of the abstraction level hierarchy. They provide a number of built-in abstractions that can be used to build programs. Moreover, they also provide ways to define new abstractions. For example, it is possible to define new functions, data structures and classes. However, these standard ways of introducing new abstractions are often insufficient. In these cases, it is feasible to extend the language itself, and thus to use the method of metalinguistic abstraction [3].

Conception of programs as multiple levels of abstraction can be considered from a language perspective, which is a basic idea of Language-Oriented Programming [4], [5]. In this methodology, the first step of program design is definition of high-level domain-specific language suitable for solving a specific problem. Next, the program itself is implemented using the new language which is built upon the existing (less abstract) language. From this point of view, each level of abstraction is represented by a language, where each language is defined using a lower level language.

2. MOTIVATION

Let us consider two pieces of pseudo-code expressing transformation of the array values:

```
output = new Array();
for (int i = 0; i < input.size; i++){
  a = input[i];
  output[i] = f(a) * 5 + 3;
}

squares = new Array();
for (int i = 0; i < numbers.size; i++){
  b = numbers[i];
  squares[i] = b ^ 2;
}
```

Both pieces of pseudo-code take all the values of arrays *input* and *numbers*, and transform them according to the appropriate calculations. The first pseudo-code uses function $f()$, multiplication and addition; the second pseudo-code uses power. All the final values are then stored in arrays *output* and *squares*.

As both pieces of pseudo-code are very similar, replacement of the repeated structures might be convenient. First, let us consider a new pseudo-code, applicable to both examples:

```
<<output>> = new Array();
for (int i = 0; i < <<input>>.size; i++){
  x = <<input[i]>>;
  <<output[i]>> = <<op x>>;
}
```

Where:

- `<<input>>` can be considered as a variable replaceable by arrays *input* and *numbers*;
- `<<output>>` represents a variable for arrays *output* and *squares*; and
- `<<op x>>` represents a variable for the calculations: $f(x) \times 5 + 3$ and x^2.

As the new pseudo-code is applicable to both examples, it can be treated as a *pattern*.

Abstraction has one simple goal in mind: To replace repeated code structures in order to increase expression abilities of the language. For the discussed examples, the identified pattern might be reduced and simplified with a new function *map* (inspired by functional programming):

```
<<output>> = map (<<input>>,<<op x>>);
```

Where *map* can be considered as an abstraction to the identified pattern, representing the whole structure of the cycle with the appropriate parameters. For the two examples, it is now possible to use new, more abstract pseudo-code:

```
output = map(input, (\x -> f(x) * 5 + 3));
squares = map(numbers, (\x -> x ^ 2);
```

This approach makes the code much shorter, and thus less prone to errors.

Several implications arise according to the mentioned considerations:

- If it is possible to recognize language structures within a source code, then it is possible to identify recurring structures as well.
- If there is a large group of source code belonging to the same application domain, then it is possible to identify plenty of recurring structures within the domain.
- If frequently repeated structures are abstracted into new ones, then it is feasible to form a new language dialect.
- If the new language structures are named by concepts of the appropriate aplication domain, then the resultant dialect is domain-specific.
- If a programmer is able to write short codes in concepts of the appropriate aplication domain instead of long codes in concepts of the general-purpose language, then his work might become much more effective.

Moreover, analysis of the current state within application of programming languages proved that along with system development in various application areas, there is a demand for the following language features [6], [7]:

- Increasing level of abstraction when expressing complex issues
- Increasing expression ability of a language, and thus effectiveness of its application
- Specialization of languages on specific domains of use
- Increasing flexibility when using a language in other domains

Considering importance of the abstraction concept in programming, there are a lot of open questions remaining, particularly regarding automatic analysis and introduction of abstraction. Therefore, in this article we will try to find answers to the following questions:

- Can effect of abstraction be measured?
- How can increase of abstraction be automated?

3. PROPOSAL

To propose a solution for automatized introduction of new language abstractions based on patterns found in source code the problem of recurring pattern recognition should be solved. Manual analysis of code may be a hard and tedious task. However, a tool for automatic pattern recognition can greatly help in this task. Moreover, recognition needs to be done at the level of program syntax.

The term of program pattern means code fragment extracted from a set of sample programs that have equivalent syntactic, and hence, also semantic structure. Patterns can also contain parts that are different in each program. These parts can be called syntactic variables.

Expressiveness of a language can be improved by the recognition of program patterns. Moreover, it allows more natural and straight-forward expression of programs. This approach can also be useful for development of domain-specific dialects of programming languages.

In order to implement this transition from general-purpose language (GPL) to its domain-specific dialect, it is necessary to reflect the fundamental differences between the domain-specific dialect and the corresponding GPL. The main differences lie in the following:

- Focus on a particular domain
- Use of concepts from a domain
- Higher abstraction

To achieve connection with a particular domain and shift towards domain specificity, it is suitable to analyze existing programs (or program fragments) written in the GPL solving various problems from the domain. On the basis of this analysis, a shift from GPL to domain-specific dialect can be achieved, overcoming the mentioned differences as follows:

- *Domain specificity* — DSL is aimed at solving problems of a particular domain and consists of structures and notations associated with the domain. Thus, it is possible to identify linguistic structures that are not used in programs, addressing problems of the domain.
- *Using concepts from the domain* — DSL uses concepts of a problem domain and defines relations among them. Thus, it is essential to find and identify domain-specific constructs that are repetitive in particular programs.
- *Higher abstraction* — GPLs are intended to solve various problems, consequently they contain only general implementations and abstraction of lower levels. They are used to create solution to a specific problem. On the other hand, DSLs are dedicated to particular domains, thus containing specific solutions and implementations in the form of a higher level of abstraction. Therefore, during the analysis, patterns recurring in individual programs have been searched for, so it was possible to unify and create higher level abstractions.

In pursuance of these facts, implementation of a domain-specific dialect from the base language consists of two parts:

a) Introducing new syntactic elements for abstractions used in the domain — *Language extension*

b) Removing syntactic elements not used (and thus not needed) in programs for the domain — *Language reduction*

4. MEASUREMENT OF ABSTRACTION

To measure the effect of abstraction, it is necessary to have an example of an abstract construct. For the purpose of this article, list comprehension was chosen.

List comprehension (or set abstraction) is a powerful construct of the Haskell programming language that enables the use of notation equivalent to Zermelo-Fraenkel set notation. It is a good example of abstraction because it provides much more abstract notation for list manipulation compared to usage of other list manipulation operations. At the same time, every list comprehension expression can be translated into the form with lower level of abstraction.

4.1. List Comprehension

List comprehension LC can be defined as an expression $[\, E \mid Q_1, \ldots, Q_n \,]$ with syntax presented in Fig. 2 [8]. If the E expression is of type T, LC is then of type $[T]$.

$$
\begin{array}{lcl}
LC & ::= & [\, E \mid QS \,] \\
QS & ::= & Q_1, \ldots, Q_n, \text{for } n \geq 0 \\
Q & ::= & G \mid F \\
G & ::= & p \texttt{ <- } L \\
p & ::= & pattern
\end{array}
$$

Fig. 2 Syntax of list comprehension [8]

LC contains a list QS of qualifiers, separated by commas. Each qualifier can be either generator G or filter F. Filter F is a logical expression, and generator G produces patterns p of list L. If $p : T_p$, then $L : [T_p]$, where pattern p can be a variable or a constant of product type (e.g. tuple).

Translation of list comprehension is defined by translation scheme $\mathscr{C}$ according to Fig. 3 [8]. Except for the lambda abstraction, application of `if` operation is applied, expressed as `if` $b\ e_T\ e_F$. In Haskell language, it is possible to represent it through expression of extended lambda language `if` b `then` e_T `else` e_F.

In the translation scheme, qs is a list of qualifiers and function h is as follows:

```
h            ::  (a -> [b]) -> [a] -> [b]
h f []       =   []
h f (x:xs)   =   f x ++ h f xs
```

It is possible to prove, that:

```
h f = concat .  (map f)
```

By expressing a simple list comprehension through extended lambda language according to the translation scheme, it is possible to practically ascertain the correctness of implementation on the basis of this scheme.

Theoretical analysis of the scheme accuracy based on `concat` and `map` functions might be simpler than analysis based on optimized function `h`. E.g. right side of equation (4) in the translation scheme:

$$\texttt{h}\ (\ \lambda p.\mathscr{C}\ [\![\ [\, E \mid qs \,]\ ]\!]\ )\ (\ \mathscr{E}\ [\![\, L \,]\!]\ )$$

can be expressed equivalently as:

$$\texttt{concat}\ (\texttt{map}\ (\ \lambda p.\mathscr{C}\ [\![\ [\, E \mid qs \,]\ ]\!]\ )\ (\ \mathscr{E}\ [\![\, L \,]\!]\ ))$$

because of the equation:

```
h f xs = (concat .  (map f)) xs
```

and thus the following equation is true as well:

```
h f xs = concat (map f xs)
```

4.2. Methodology of Measurement

Let us consider four list comprehension expressions:

```
f1 = [ x | x<-[1,2,3] ]
f2 = [ (x,y) | x<-[1,2,3], y<-[10,20,30] ]
f3 = [ (x,y) | x<-[1,2,3], y<-[10,20,30],
 x>=2 ]
f4 = [ (x,y) | x<-[1,2,3], y<-[10,20,30],
 x>=2, y<=25 ]
```

ISSN 1335-8243 (print) © 2012 FEI TUKE
www.aei.tuke.sk

ISSN 1338-3957 (online)
www.versita.com/aei

$$
\begin{array}{lcll}
\mathscr{E}[\![\,[\,E \mid qs\,]\,]\!] & = & \mathscr{C}[\![\,[\,E \mid qs\,]\,]\!] & \\
 & & & \\
\mathscr{C}[\![\,[\,E \mid\,]\,]\!] & = & [\,\mathscr{E}[\![\,E\,]\!]\,] & (1)\\
\mathscr{C}[\![\,[\,E \mid F,\, qs\,]\,]\!] & = & \texttt{if}\,(\,\mathscr{E}[\![\,F\,]\!]\,)\,(\,\mathscr{C}[\![\,[\,E \mid qs\,]\,]\!]\,)\,(\,[\,]\,) & (2)\\
\mathscr{C}[\![\,[\,E \mid F\,]\,]\!] & = & \texttt{if}\,(\,\mathscr{E}[\![\,F\,]\!]\,)\,(\,\mathscr{C}[\![\,[\,E \mid\,]\,]\!]\,)\,(\,[\,]\,) & (3)\\
\mathscr{C}[\![\,[\,E \mid p \leftarrow L,\, qs\,]\,]\!] & = & \texttt{h}\,(\,\lambda p.\mathscr{C}[\![\,[\,E \mid qs\,]\,]\!]\,)\,(\,\mathscr{E}[\![\,L\,]\!]\,) & (4)\\
\mathscr{C}[\![\,[\,E \mid p \leftarrow L\,]\,]\!] & = & \texttt{h}\,(\,\lambda p.\mathscr{C}[\![\,[\,E \mid\,]\,]\!]\,)\,(\,\mathscr{E}[\![\,L\,]\!]\,) & (5)
\end{array}
$$

Fig. 3 Translation scheme of list comprehension [8]

These expressions can be translated into less abstract forms using translation scheme described in section 4.1. For example, translated version of `f1` is as follows:

```
l1 = concat (map (\x -> [x]) [1,2,3])
```

This might look quite simple, however, translated version of `f4` is more complex:

```
l4 = concat (map (\x ->
     concat (map (\y ->
      if x>=2 then (if y <=25 then [(x,y)]
                              else [])
              else [])
      [10,20,30]))
     [1,2,3])
```

These programs are examples of how to express the same meaning using different levels of abstraction. The abstraction used in this case is language based abstraction — it is achieved by additional language construct (so called "syntactic sugar").

Using the Haskell syntax analysis, derivation trees of these programs have been produced. In Fig. 4, it is possible to compare derivation tree of list comprehension expressed by `f4` to derivation tree of its translation (less abstract version) expressed by `l4`.

The more complex is the list comprehension, the wider is its derivation tree (more nodes). However, the tree itself remains fairly comprehensive.

On the other hand, each derivation tree representing less abstract expression of the same list gathers not only new nodes, but it is also compounded by transitions between nodes, thereby extending the tree into its depth, and therefore reducing efficiency of the program result production.

Let us call programs defining list comprehensions M_1, M_2, M_3, and M_4. We have detailed (less abstract) representations of these programs received using the translation scheme described in Section 4.1 as well. Let us call them D_1, D_2, D_3, and D_4. For each program P, it is possible to create its derivation tree $T(P)$ based on the language syntax.

Let $c(P)$ be the length of program code — the number of characters excluding white spaces. Then it is possible to define ratio of abstraction of target (program code) as $Z_i = \frac{c(D_i)}{c(M_i)}$.

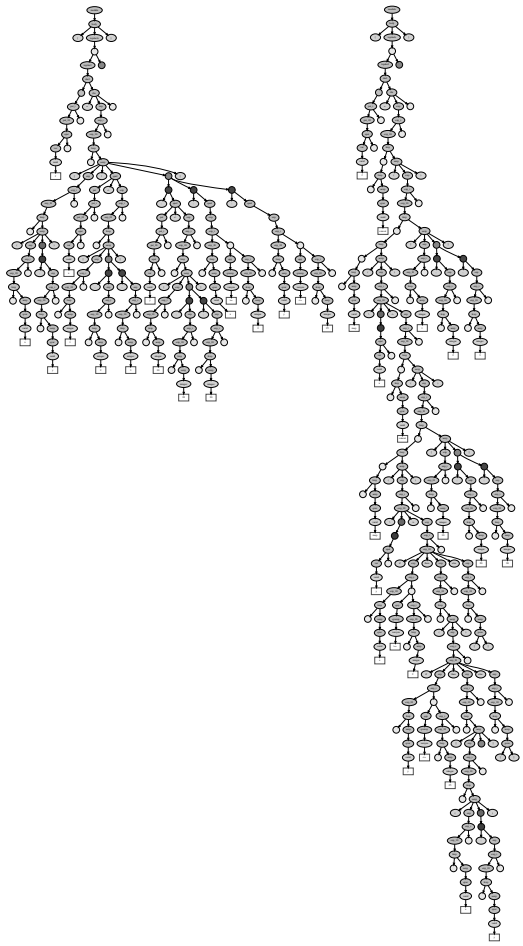

Fig. 4 Comparison of derivation trees of `f4` and `l4`

Let $n(G) = |V(G)|$ be the order of graph G — number of graph nodes. Then it is possible to define ratio of abstraction of derivation as $T_i = \frac{n(T(D_i))}{n(T(M_i))}$.

Analyzing these ratios for a number of programs, impact on program depth and size of derivation tree is obvious. Relation between these parameters is visible as well.

Table 1 contains results of the measurement for experimental programs. This simple experiment implies that abstraction has greater impact on target form of the program (source code) than on its derivation because relative change of length is greater in target form ($Z_i > T_i$).

Table 1 Results of the abstraction experiments

i	$c(M_i)$	$c(D_i)$	$n(M_i)$	$n(D_i)$	Z_i	T_i
1	16	29	88	113	1.81	1.28
2	34	60	159	208	1.76	1.31
3	39	76	181	244	1.95	1.35
4	45	95	203	289	2.11	1.42

This means less effort on more abstract production, providing higher level of target abstraction (in our case, target is the source code form) implying that prevention of low levels of abstraction might be convenient. Thus, higher level of abstraction increases the level of transparency and reliability of programs.

5. EXPERIMENTS ON LANGUAGE PATTERNS

For experimental purposes, Haskell 98 was chosen as a language for the analysis. To get a proper knowledge about language constructs and syntactic structure of the analyzed programs, a complex set of tools has been developed. As a result of one program analysis, derivation tree is produced, consisting of the used Haskell grammar rules [9].

Architecture of syntax analysis consists of two parts:

- *Generating infrastructure*
- *Analyzing infrastructure*

The goal of generating infrastructure is to prepare tools being used during the analysis, and the analyzing infrastructure contains lexical analyser (lexer) and parser, intended for analysis of specific programs into lexical units, then processing them into derivation trees. Derivation trees have been produced for further process to retrieve statistical data on the programs, and to recognize common language patterns.

5.1. Code Statistics

Using the tools developed for these experiments, it was possible to compute several interesting statistics based on a set of about 300 Haskell sample programs. As a result of one program analysis, its derivation tree is provided according to the language grammar. Resulting derivation tree consists of terminal and nonterminal symbols, where terminal symbols represent leaves of the tree. The derivation tree also contains helper nodes corresponding to EBNF features like repetition or optional elements.

One of the parameters that may be investigated is a relative occurrence of symbols in derivation trees. Relative occurrence of symbol in a program is defined as $r_{sym} = \frac{n_{sym}}{N}$, where n_{sym} means a number of occurrences of the *sym* symbol in the derivation tree and N represents a number of all symbols/nodes of the derivation tree.

Table 2 represents 10 most frequent occurrences of particular symbols in all programs of our sample. As it might have been expected, variable names and expressions have the greatest frequency. However, some symbols even did not occur in any of our sample programs, like `default`, `fbind`, `fpat` and `gdpat`.

Table 2 Proportion number of 10 most frequent symbol occurrences

Symbol	Occurrence	Symbol	Occurrence
varid	0,093855	exp_i	0,049428
aexp	0,092660	exp	0,044523
fexp	0,092660	var	0,037632
exp_10	0,063059	apat	0,033349
qvar	0,051154	conid	0,026202

It is possible to provide similar statistics for specially selected sample of programs within a specific domain. This might show which language elements are used in programs of a particular domain and which elements can be omitted from the domain-specific dialect. Moreover, statistical analysis can also be used to partition sample programs into groups based on usage of language elements.

5.2. Pattern Recognition

To recognize syntactic patterns in a program or a set of programs, it is important to decide which parts of the analyzed programs may be considered similar. The simplest possibility is to consider only the equal trees. However, this approach is exceedingly limiting. Trees can be considered similar if their structure is the same except for the attributes of terminal symbols (approach that has been chosen).

Another approach is to allow differences in whole subtrees rooted in the same type node. This would allow more complex syntactic variables, but it is harder to implement.

To find patterns in the program derivation tree, a simple algorithm can be used, based on the function $findPatterns$ defined below:

$parents \leftarrow allParents(elements)$
$groups \leftarrow findGroups(parents)$
if *groups* is empty **then**
  return $[elements]$
else
  for all $group \in groups$ **do**
    Add $findPatterns(group)$ to $foundGroups$
  end for
  return $mergeGroups(foundGroups)$
end if

Function $findPatterns$ takes a list of the tree elements and recursively examines their parents to find a set of groups of subtrees that have a similar structure. It uses helper functions where *allParents* returns a set of parents of all tree elements in a group. Given a set of tree elements, $findGroups$ returns list of groups of elements with similar subtrees. *mergeGroups* merges list of group lists into a single list.

To initiate the algorithm, the $findPatterns$ function is called on terminal symbols of the tree. Then it tries to walk up to the root of the tree while it can find groups of subtrees with similar structure. List of subtree groups is a result of the algorithm, where each group corresponds to a found pattern and contains all occurrences of the pattern.

Let us look at a simple example program defining function *eval* evaluating expressions defined using derived abstract syntax tree. Derivation tree of this program is represented in Fig. 5.

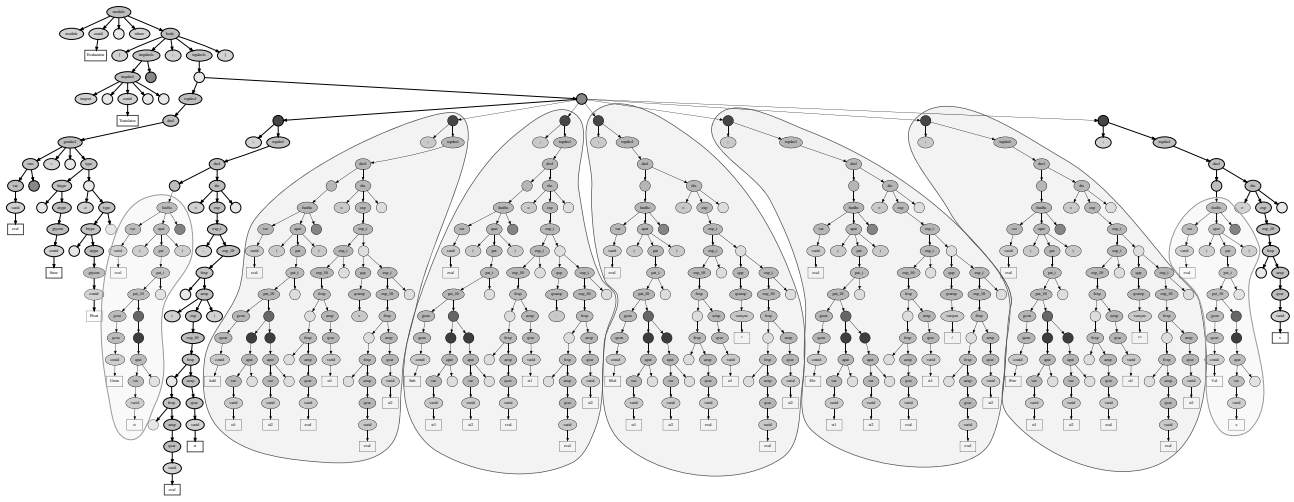

Fig. 5 Example of a derivation tree program with recognized patterns

```
module Evaluation where
import Translator

eval :: Stree -> Float
eval (Umin st)     = -(eval st)
eval (Add st1 st2) = eval st1 + eval st2
eval (Sub st1 st2) = eval st1 - eval st2
eval (Mul st1 st2) = eval st1 * eval st2
eval (Div st1 st2) = eval st1 / eval st2
eval (Pow st1 st2) = eval st1 ** eval st2
eval (Val x)       = x
```

Using the described method, it is possible to find several recurring patterns in this program (see Fig. 5). The most important are:

- `eval (`α `st1 st2) = eval st1` β `eval st2`
- `eval (`α β`)`

Greek letters in the patterns represent syntactic variables that can be replaced with concrete syntactic elements. Other recognized patterns are too small to be mentioned.

6. CONCLUSION AND FUTURE WORK

In this article we have shown that abstraction in programming languages has great effect on programs. This effect was analyzed on the process of source code derivation based on language syntax. Experiments were performed via Haskell syntax analysis, gathering needed information from Haskell programs and retrieving their derivation trees.

List comprehension and its translation corresponds directly to various levels of abstraction in the programs, and the produced derivation trees reflect that these levels of abstraction have strong impact on program derivation process. Analysis of target abstraction ratio and derivation abstraction ratio corresponds to conclusions proclaimed after producing and comparing derivation trees — that relative change of length is greater in target form than in the source form.

Reduction of the base language is very important and should not be overlooked. By reducing unneeded syntactic elements, the language becomes easier to learn. It also decreases possibility of errors that may result from accidental usage of wrong language elements. Moreover, reduction of unneeded elements can also allow syntax simplification of the rest of the language.

To make more significant conclusions, it is necessary to perform experiments on greater set of programs. One helpful indication is that slight variation of list comprehension within eight functions yields plausible results. This supports the idea that is already known from functional programming regarding the fact that a language should be as simple as possible and, at the same time, it may be able to express a solution for any problem in a given problem area.

Further research will focus on methods of flexible language restructuring, based on a new form of language grammars. Then the aim should be resilient language adaptation to another application domain that may contribute to construction or specialization of domain-specific languages as well.

We have also presented experiments that made it possible to accomplish pattern recognition in program code, with perspective of new dialect development, both general-purpose and domain-specific. The term of program patterns was used for syntactically and semantically equal program fragments occurring in a set of program samples.

As shown in this article, having a grammar of a language as well as a set of program samples, we are able to evaluate the usage frequency of symbols (concepts in the language).

This may be interesting from the perspective of language benchmarking, the goal of which is to reduce the amount of redundant constructs. Thus, further research also involves an extension for processing a whole set of programs. Another usage might be extension of a language based on the needs of programmers [10]. It may allow adding new constructs to the language corresponding to repeated code fragments.

However, upon the presented results, the most significant is the contribution to automated software evolution. Clearly, this would mean to shift from a language analysis to language abstraction, associating concepts to formal language constructs [11], and formalizing them by means of these associations. In this way, we expect to integrate programming and modeling, associating general purpose and domain-specific languages [12], [13], as well as to perform a qualitative move from an automatic roundtrip engineering [14], [15] to the automated roundtrip software evolution, that is understood as the software development without any affects of a human.

Therefore, experiments performed in this article outcome additional experiments with new conception and features of language patterns, meaning the rise of language expression ability, covering current paradigms. Next, the research will also focus on new methods of composition (or combination) of programs, based on new conception of language patterns. This new approach would also mean theoretical contribution to language grammars based on new pattern conception, with the aim of flexibility increase in specialization of patterns in particular fields of application.

ACKNOWLEDGEMENT

This work was supported by VEGA Grant No. 1/0305/11 Co-evolution of the artifacts written in domain-specific languages driven by language evolution.

REFERENCES

[1] BROOKS, F.P.: No Silver Bullet: Essence and Accidents of Software Engineering, *IEEE Computer*, Los Alamitos, Vol. 20, pp. 10–19, 1987, ISSN 0018-9162.

[2] ABELSON, H. – SUSSMAN, G.J.: Lisp: A Language for Stratified Design, *Massachusetts Institute of Technology*, Cambridge MA, 1987.

[3] ABELSON, H. – SUSSMAN, G.J.: Structure and Interpretation of Computer Programs, 2^{nd} edition, *The MIT Press*, 1996, ISBN 0262011530.

[4] WARD, M.P.: Language-Oriented Programming, *Software - Concepts and Tools*, Vol. 15, No. 4, 1994.

[5] DMITRIEV, S.: Language oriented programming: The next programming paradigm, *JetBrains*, 2004.

[6] ASTAPOV, D.: Using Haskell with the support of business-critical information systems, *Practice of Functional Programming (in Russian)*, Vol. 2, 2009, ISSN 2075-8456.

[7] OTT, A.: Using Scheme in the Development of "Dozor-Jet" family of products, *Practice of Functional Programming (in Russian)*, Vol. 2, 2009, ISSN 2075-8456.

[8] KOLLÁR, J.: Functional Programming (In Slovak), *Elfa*, 2009, ISBN 978-80-8086-116-2.

[9] PEYTON JONES, S.: Haskell 98 Language and Libraries – The Revised Report, *Cambridge University Press*, Cambridge England, 2003.

[10] STEELE, G.L.: Growing a Language, *Higher-Order and Symbolic Computation*, Springer Netherlands, Vol. 12, pp. 221–236, 1999, ISSN 1388-3690.

[11] PORUBÄN, J. – VÁCLAVÍK, P.: Extensible Language Independent Source Code Refactoring, *AEI '2008: International Conference on Applied Electrical Engineering and Informatics*, Athens, pp. 58–63, 2008.

[12] SABO, M. – PORUBÄN, J.: Preserving Design Patterns using Source Code Annotations, *Journal of Computer Science and Control Systems*, Vol. 2, No. 1, pp. 53–56, 2009.

[13] LUKOVIĆ, I. et al.: An approach to developing complex database schemas using form types, *Software Practice & Experience*, John Wiley & Sons, Vol. 37, No. 15, pp. 1621–1656, 2007.

[14] ASSMANN, U.: Automatic roundtrip engineering, *Electronic Notes in Theoretical Computer Science*, Elsevier, Vol. 82, No. 5, pp. 33–41, 2003.

[15] LOHMANN, C. et al.: Applying triple graph grammars for pattern-based workflow model transformations, *Journal of Object Technology*, Vol. 6, No. 9, pp. 253–273, 2007.

Received April 19, 2012, accepted June 8, 2012

BIOGRAPHIES

Ján Kollár if Full Professor of Informatics at Department of Computers and Informatics, Technical University of Košice, Slovakia. He receiver 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 Computers 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.

Emília Pietriková is PhD student at Department of Computers and Informatics, Faculty of Electrical Engineering and Informatics, Technical University of Košice, Slovakia. She received her MSc in Informatics in 2010 at Technical University of Košice. The subject of her research is metaprogramming, programming paradigms, and exploiting functional paradigm in system evolution.

Sergej Chodarev is PhD student at Department of Computers and Informatics of Faculty of Electrical Engineering and Informatics at Technical university of Košice, Slovakia. He received his MSc in Computer Science in 2009. The subject of his research is domain-specific languages, metaprogramming and programming paradigms.