Towards Automated Program Abstraction and Language Enrichment*

Sergej Chodarev, Emília Pietriková, and Ján Kollár

Department of Computers and Informatics
Technical University of Košice
Letná 9, 042 00 Košice, Slovakia
{Sergej.Chodarev,Emilia.Pietrikova,Jan.Kollar}@tuke.sk

Abstract

This paper focuses on the presentation of a method for automated raise of programming language abstraction level. The base concept for the approach is a code pattern – recurring structure in program code. In contrast to design patterns it has a specific representation at a code level and thus can be parameterized and replaced by a new language element. In the article two algorithms for automated recognition of patterns in samples of programs are described and examined. The paper also presents an approach for language extension based on the found patterns. It is based on an interactive communication with the programming environment, where recognized patterns are suggested to a programmer and can be injected into the language in a form of new elements. Conducted experiments are evaluated in regard to the future perspective and contributions.

1998 ACM Subject Classification D.3.2 Language Classifications – Extensible languages, D.2.6 Programming Environments

Keywords and phrases Abstraction, code patterns, language extension, projectional editing

Digital Object Identifier 10.4230/OASIcs.SLATE.2013.51

1 Introduction

One of the matters that make software development hard is the complexity. It is caused by the inherent complexity of the problems that developed systems need to solve and also by the need to comply to existing norms, interfaces and protocols [5]. 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 hiding the 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 defining new, more abstract concepts and language constructions.

Abstractions are usually organized into several levels, where each level is built on the abstractions provided by the level below it. This practice is called stratified [1] or layered design. Provided that lower levels are already in place, it is possible to concentrate on problem solution that can be expressed in high level terms relevant to the domain of solved problem.

* This work was supported by project VEGA 1/0341/13 *Principles and methods of automated abstraction of computer languages and software development based on the semantic enrichment caused by communication.*

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 can also provide ways to define new abstractions. For example, it is possible to define new functions, data structures and classes. This allows distinguishing two ways of how new abstraction can be defined [21]:

1. *Linguistic* abstraction, with abstractions built into the language, what is typical for DSLs (Domain-Specific Languages),
2. *In-language* abstraction, with abstractions expressed by concepts available in the language, typical for GPLs (General-Purpose Languages).

Linguistic abstraction makes new abstract concepts part of the language itself. Definition of the concepts becomes a part of language translator or interpreter. Integration of abstractions into a language provides several advantages. First of all it provides a possibility to use the most appropriate notation. The use of appropriate linguistic abstraction instead of describing the same program using more general concepts can also provide additional semantic information about programmer intents for language processor, thus allowing optimizations to be performed without the need to reverse engineer the semantics. This is because no details irrelevant to the model need to be expressed, increasing conciseness and avoiding over-specification [21].

Compared to the linguistic, in-language abstraction constitutes achieving conciseness by providing facilities allowing users to define new (non-linguistic) abstractions in programs [21], including procedures or functions, and higher-order functions or monads. This way, abstraction can be provided but no declarativeness, in the sense of direct mapping of the program concepts to the model semantics. Thus, in-language abstraction is more flexible as users can build only those abstractions they actually need.

Linguistic abstraction is a basic element of Language-Oriented Programming [22, 6]. In this methodology, the first step of program design is a definition of high-level domain-specific language suitable for solving a specific problem[1]. 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.

In this paper we propose new approach for the introduction of linguistic abstractions based on automated pattern recognition in program code. Section 2 provides a motivation for our work. In section 3 the proposal of programming environment supporting language enrichment is presented. Section 4 describes our experiments with two methods for pattern recognition. Finally section 5 concludes the paper and proposes ideas for future research.

2 Motivation

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

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

[1] The process of solving a problem by designing new language first is itself also called metalinguistic abstraction [2]

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

Both pieces of pseudo-code take all the values of arrays *data* 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 function. All the final values are then stored in arrays *results* and *squares* respectively.

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 *data* and *numbers*;
- `«output»` represents a variable for arrays *results* and *squares*; and
- `«op x»` represents a variable for the calculations: $f(x) \times 5 + 3$ and x^2.

As it is possible to apply this new pseudo-code to both examples, it can be regarded as a *pattern*, which is repeated in the examples.

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 construct *map* (inspired by functional programming):

```
«output» = map («input», «op x»);
```

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

```
results = map(data, (\x -> f(x) * 5 + 3));
squares = map(numbers, (\x -> pow(x, 2));
```

This approach enables the program code to be much shorter, 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 the new ones, then it is feasible to form a new language dialect.
- If the new language structures are named by concepts of the appropriate application domain, then the resultant dialect is domain-specific.
- If a programmer is able to write short codes in concepts of the appropriate application domain instead of long codes in concepts of the general-purpose language, then his work might become much more effective.

- If the programming environment would provide help with definition of new abstractions, then there is a higher chance abstractions would be actually used.

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 [4, 14]:

- 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 the following sections we will try to find answer (or more answers) to the following main question: *How can increase of abstraction be automated?*

Our approach to this task is based on resolving two basic problems concerning tool support for automated program abstraction and language enrichment:

1. Recognizing recurring patterns in program code.
2. Finding a way to inject identified patterns into a language as new constructs.

3 Proposal

We decided to base our approach to program abstraction on the concept of *patterns* – recurring structures in program codes. The conceptual scheme of the proposal can be seen in Fig. 1.

To propose a solution for automated introduction of new language abstractions based on patterns found in source code the problem of recurring pattern recognition should be resolved. Manual analysis of code may be a hard and tedious task. However, a tool for automatic pattern recognition can greatly help in this task.

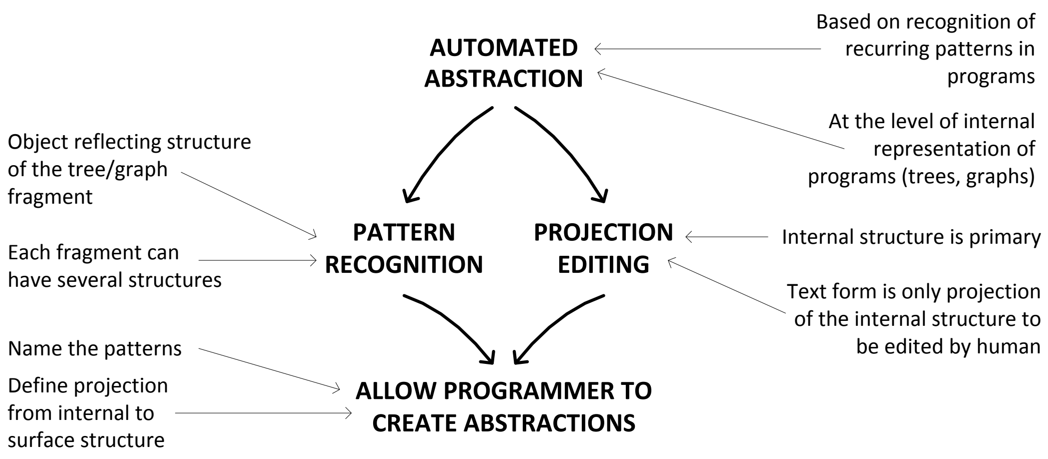
AUTOMATED ABSTRACTION
Based on recognition of recurring patterns in programs
At the level of internal representation of programs (trees, graphs)
Object reflecting structure of the tree/graph fragment
Each fragment can have several structures
PATTERN RECOGNITION
PROJECTION EDITING
Internal structure is primary
Text form is only projection of the internal structure to be edited by human
Name the patterns
Define projection from internal to surface structure
ALLOW PROGRAMMER TO CREATE ABSTRACTIONS

Figure 1 Proposal conceptual scheme.

3.1 Pattern Recognition

Pattern for our purposes is a recurring structure in program code. This structure can be expressed by fragment of a program with parts that may be different, replaced by *pattern variables*. This concept is different from *design patterns* [9] which describe recurring patterns and their usage on higher level. On the other hand, we are currently concentrating on patterns of a smaller scale.

Patterns would be recognized at the level of language abstract syntax. The abstract syntax tree or graph of a program contains all important information about structure of its code without syntactic details.

We have developed two approaches to this task:

1. Pattern recognition by comparing
2. Pattern recognition by collecting

Recognition by comparing is based on traversing trees representing programs from leaves up and comparing the subtrees to find groups of subtrees with the same structure.

The second approach that we propose is based on collection of abstracted structural schemas of program fragments. They can be obtained by replacing parts of the fragment by variables. Each fragment of code may be described by several structures of different level of detail. If such structural schemas are identified and stored with references to program fragments that contain them, it is possible to analyze frequency of their usage and by this way identify possible pattern.

3.2 Language Enrichment

Since recognized patterns represent recurring structures found in programs, they also present potential extensions of the language. To inject a pattern into the language there is a need to name it and define its syntactic or surface structure that would be used to represent it in program code.

Enrichment of language syntax and semantics directly by its users, though, is rarely allowed. In most cases it requires modification of the original language implementation, since a composition of the language with new elements is needed. Doing so with traditional textual language processed using a parser usually generated by some parser generator according to the grammar specification, it may result in several problems caused by possible ambiguity of the resulting grammar. Partially, this is caused by the fact, that grammar subclasses supported by common parser generators are not closed under composition [11]. Moreover, to allow composition and extension of a language without modification of the original implementation, it requires special tool support [7] and knowledge in the field of language development.

A possible way to solve the problem of language composition is a transition to *concept composition*. This means that instead of composing languages and their grammar rules, only concepts in a single base language are composed. This requires lowering the role of language grammar and is possible to be achieved at least in two ways:

1. Using single syntax for composed languages.
2. Using projectional editing.

In our case, the first way means not to use special syntax for injected patterns. Patterns would only be named and one of the shapes predefined in the language syntax would be assigned to them. This is similar to the definition of a function – it is assigned a name and standard syntax of function call or operator application. Another example is extension of Lisp and its dialects using macros that are based on the uniform syntax of S-expressions [10],

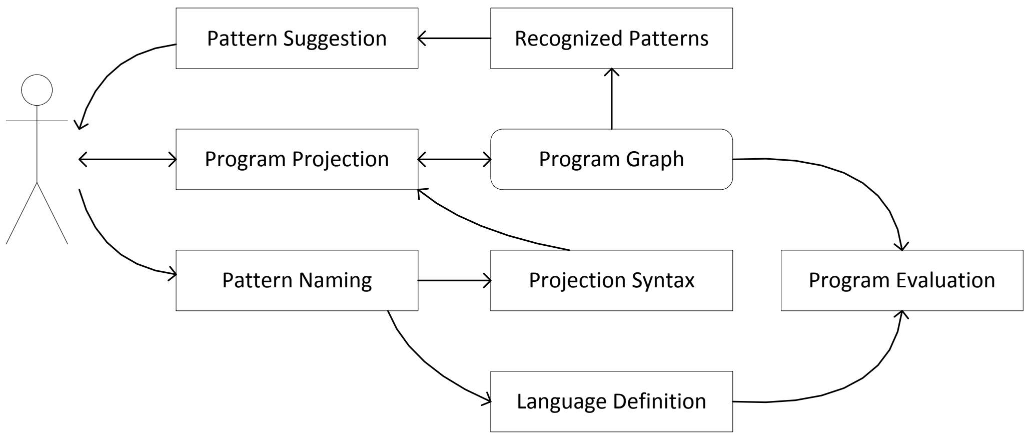

Figure 2 Architecture of language enrichment environment.

or definition of XML based language. Disadvantage of such an approach is, indeed, low flexibility of choosing an appropriate notation.

On the other hand, projectional editing keeps different notations for languages on the surface, while using unified representation internally. The syntax becomes only a matter of projection and actual information of a program (code) is stored in some different form, not visible for language user.

Projectional editing is used by some language workbenches, for example JetBrains MPS [6] or Intentional Workbench [19]. They use internal graph-based representation as main form of a program. Editable form is only a projection of internal form [8]. When a user is issuing editing commands at the projection, the internal structure is modified and the projection is updated accordingly. This allows different types of editable representation in addition to the textual, for example graphical or table-based.

In this way, elements of languages developed using a language workbench are actually only concepts of an internal representation language (which is usually not textual). This means that in this case the composition of languages corresponds to the composition of concepts inside a single language. Textual composition is only its projection which is not required to be unambiguous.

3.3 Concept of Language Enrichment Environment

Based on the considerations described above, it is possible to construct an environment for automated language enrichment based on patterns found in programs. The principles of its functionality are depicted in Fig. 2. The primary representation of the program is its abstract syntactic graph (ASG) editable through the projection. Syntax specification is used as a basis for projection and editing environment.

While program is created by a programmer, the structure of its elements is collected inside the environment to recognize the recurring patterns. Then, the environment should use the collection to suggest recognized patterns to programmer. Patterns can have various uses beside the automated abstraction. For instance, they can provide automatic completion of snippets for frequently used constructs.

If a programmer decides the pattern is semantically significant, he can name the pattern and therefore enrich the language he uses. He needs to provide specific concrete syntax for the named pattern that would be appropriate for conveying its meaning. A new syntax rule would be added to the language syntax definition and used by the projectional editor.

At the same time, named pattern becomes a part of the language structure and semantics definition. By default it expands the pattern during program evaluation or translation. Definition though can be amended by some optimization rules based on semantics of the pattern.

Introduction of new abstractions into the language can also have a negative effect. The more new constructs specific to a program are added into the language, the more is a programmer forced to learn new abstractions. Moreover, if more than one programmer is working with the same program code, and only few of them know the new abstractions, a problem may occur as they may not understand program codes of each other. This situation may occur already in the current abstraction range of various general-purpose and domain-specific languages.

Considering this, it would be appropriate if the environment would also provide reverse mode of work. The user should be allowed to switch the level of abstraction used in the code and display it in expanded form. That is, our experiments and algorithms for pattern recognition should result in two instruments available to user of the programming environment:

- Pattern contraction – pattern replacement by new syntactic element
- Pattern expansion – syntactic element replacement by its implementation through elements of lower level abstraction

For instance, if one programmer knows list comprehension construct, the other one, who does not, should by able to specify directly within the environment that he does not want to use list comprehension, or that he wants to learn their structure. Then, any list comprehension would be equivalently substituted, e.g. through map or filter function. This expansion can be even applied repeatedly as it is shown in Listing 1. On the contrary, a programmer may also use several complex structures, and then replace them through the known patterns by equivalent, shorter structures, and thus noticeably reduce the program code length.

Listing 1 Example of language element expansion.

```
squares = [x ^ 2 | x <- xs]
   ↓
squares = map (\x -> x ^ 2) xs
   ↓
squares [] = []
squares (x:xs) = (x ^ 2) : squares xs
   ↓
squares xs | null xs = []
           | otherwise = (head x ^ 2) : squares (tail xs)
```

4 Experiments

For pattern recognition, we suggest two different methods: by comparing and by collecting. For experimental purposes, the first method has been performed and examined on a large group of Haskell programs while the second method has been performed on a simple language of functions and expressions.

Pattern recognition by comparing is also a successor to another research, as it uses results and implementations acquired by experiments associated with effects of abstraction [16].

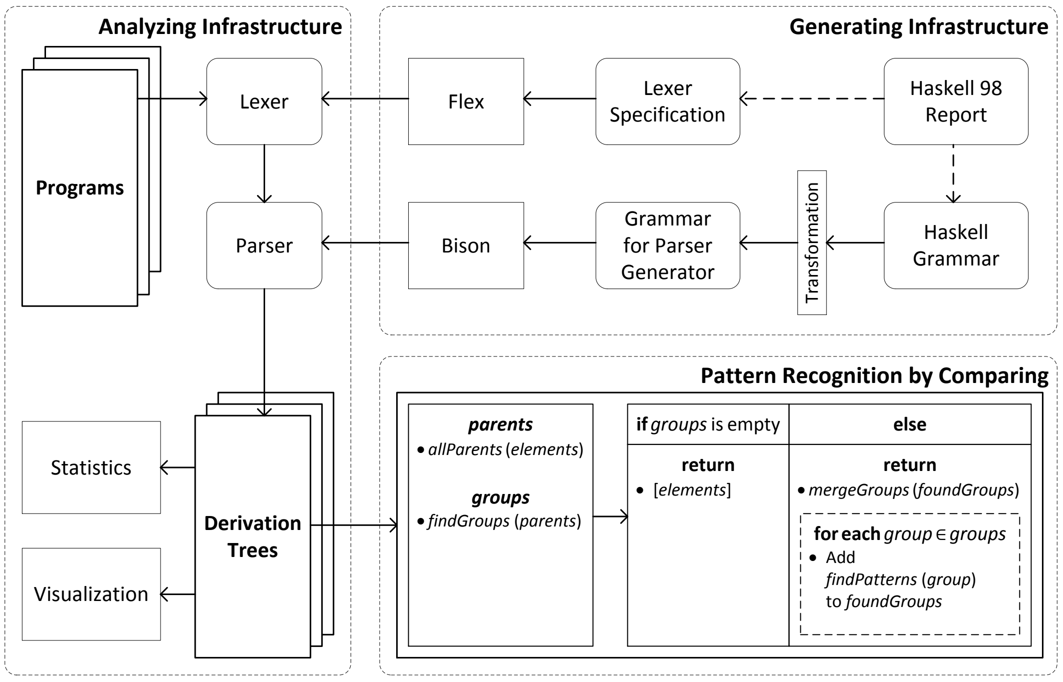

Figure 3 Architecture of Haskell syntax analyzing tools, including pattern recognition by comparing.

On the other hand, pattern recognition by collecting is based on the evaluation of the first method, reducing unnecessary implementations and extending possibilities of the pattern recognition.

4.1 Pattern Recognition by Comparing

Pattern recognition method based on comparing program fragments was developed on top of the set of tools gathering information from Haskell programs to get a proper knowledge about the used constructs in analyzed programs. As a result of the program analysis, derivation tree is produced, consisting of the used rules of Haskell grammar [15]. The architecture consists of three parts – *generating infrastructure*, *analyzing infrastructure*, and *pattern recognition* (see Fig. 3).

The goal of the generating infrastructure is to prepare lexer and parser, which are used within the analyzing infrastructure, and are developed using generative methods. Haskell grammar specification is analyzed and transformed into a form suitable as an input for lexer and parser generators. The analyzing infrastructure contains lexer and parser of Haskell programs, intended for analysis of Haskell into lexical units and then processing them into derivation trees. Derivation trees are produced in XML format and are further processed to retrieve statistical data on Haskell programs and to recognize common language patterns.

The third part of the architecture, which builds on the generating and analyzing infrastructures, is dedicated to pattern recognition by comparing. It is based on the principle of comparing different fragments of a program to find groups of similar ones. The use of derivation trees generated by the analyzing infrastructure is indeed not obligatory for the proposed algorithm. Nevertheless, generating and analyzing parts of the Haskell syntax analysis tools have already been part of other range of experiments devoted to effects of abstraction in programming languages, also published and more particularly described in [16].

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 in this experiment).

Another approach is to allow differences in whole subtrees rooted in the same type node. This should allow more complex syntactic variables and is, however, more difficult to implement using specified approach, as it requires comparisons of program fragments on different levels of the tree.

To find patterns in the program derivation tree, a simple algorithm is used, based on the function *findPatterns* defined in Listing 2 (see also Fig. 3).

Listing 2 Pseudo-code algorithm for pattern recognition by comparing.

```
parents ← allParents(elements)
groups ← findGroups(parents)

if groups is empty then
  return [elements]
else
  for all group ∈ 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 and *mergeGroups* merges the list of group lists into a single list. An important function is *findGroups*. Given a set of tree elements, it returns list of groups of elements with similar subtrees.

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 in Listing 3 defining functions *insert* and *delete* manipulating binary search trees. Derivation tree of this program is represented in Fig. 4.

Using the described method, it is possible to find several recurring patterns in this program (see Fig. 4). The most important are:

- `| x` α `y = Bin y t1 (` β `x t2 )`
- `Bin y (` α `x t1 ) t2`
- α `x ( Bin y t1 t2 )`
- α `x Nil`

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

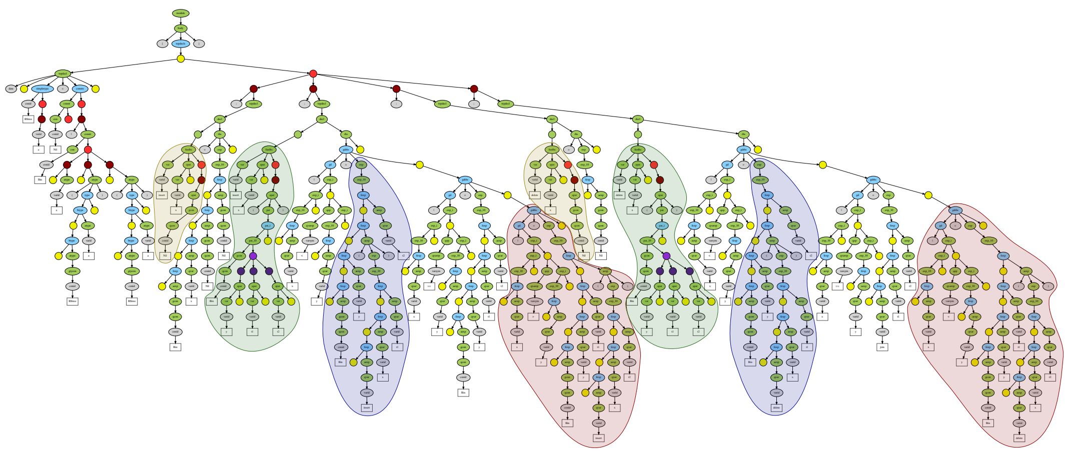

Figure 4 Example of program derivation tree with recognized patterns.

Listing 3 Example program code for pattern recognition by comparing.

```
data BStree a = Nil
    | Bin a (BStree a) (BStree a)

insert x Nil = Bin x Nil Nil
insert x (Bin y t1 t2)
    | x < y   = Bin y (insert x t1) t2
    | x == y  = Bin y t1 t2
    | x > y   = Bin y t1 (insert x t2)

delete x Nil = Nil
delete x (Bin y t1 t2)
    | x < y   = Bin y (delete x t1) t2
    | x == y  = join t1 t2
    | x < y   = Bin y t1 (delete x t2)
```

4.2 Pattern Recognition by Collecting

Another experiment has been performed to evaluate different algorithm for pattern recognition. It is based on collecting potential patterns found in a processed program. This allows evaluation of their frequency in a program and selection of patterns that may be interesting for a programmer. Basically, compared to the first method (by comparing), it is unique in its ability to recognize new patterns which differ from each other in their subtrees, not only leaves as it is in the first method. As opposed to the previous experiment, the input consists of abstract syntax trees and not derivation trees.

The main idea of the proposed algorithm is to avoid direct comparison of program fragments with each other. Instead, structure of a fragment should be described using *structural schema*. This is a data structure that reflects structure of program fragment with some details omitted. It is obvious that a fragment of program can correspond to several structural schemas which describe different parts of it and with different levels of detail.

In our case structural schema is implemented as modified abstract syntax tree with some nodes or leaves replaced by variables. Variables can match different subtrees allowing coverage of program fragments that differ whole subexpressions. Structural schemas are

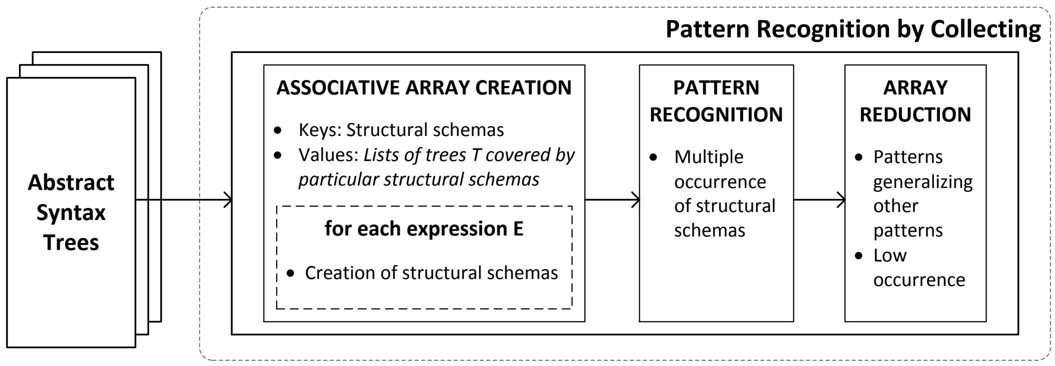
Pattern Recognition by Collecting
Abstract Syntax Trees
ASSOCIATIVE ARRAY CREATION
Keys: Structural schemas
Values: Lists of trees T covered by particular structural schemas
for each expression E
Creation of structural schemas
PATTERN RECOGNITION
Multiple occurrence of structural schemas
ARRAY REDUCTION
Patterns generalizing other patterns
Low occurrence

Figure 5 Architecture of pattern recognition by collecting tools.

actually potential patterns and therefore they have the same structure.

Fig. 5 depicts an algorithm for pattern recognition by collecting that was performed within the experiment. The input is a sequence of abstract syntax trees representing expressions of a program or collection of programs. In the first step structural schemas are derived from each expression tree. The process is outlined in Listing 4.

Listing 4 Pseudo-code algorithm for pattern recognition by collecting.

```
patterns ← empty associative array
for all expression ∈ expressions do
  subexpressions ← allSubtrees(expression)
  for all subexpression ∈ subexpressions do
    schemas ← structuralSchemas(subexpression)
    for all schema ∈ schemas do
      Add subexpression to patterns[schema]
    end for
  end for
end for
return patterns
```

The key part is the generation of structural schemas based on the fragment of program syntax tree (represented by the function *structuralSchemas* in the listing). Generated schemas actually represent possible modifications of a particular tree. By modification, we mean substitutions of the tree leaves or nodes by variables. In different schemas different combinations of nodes would be substituted.

As the result, an associative array is created of which the keys are structural schemas and values are lists of trees or subtrees covered by a particular structural schema. Within the associative array it is possible to determine multiple occurrences of the structural schemas. These schemas are important as they can be considered as patterns. Moreover, it is possible to reduce the associative array by those schemas that represent generalization of other schemas, without higher frequency.

Disadvantage of the algorithm is that it is not possible to generate and store all possible structural schemas for larger program fragments. This limitation can by overcame using different approaches:

1. Collecting only schemas for small program expressions.
2. Limit the depth of subtrees that are taken into account while generating structural schemas. All details below the specified threshold would be always replaced by variables.

While the first approach would limit possibly recognized patterns significantly, the second one may still be able to recognize a lot of useful patterns.

This algorithm has been evaluated within an experiment based on a simple language of functions and expressions. To simplify the development and make the relation between internal structure and concrete syntax more direct, S-expressions were used in the experiment.

For instance, if the code in Listing 5 is used as an input for the algorithm, it successfully recognizes a pattern in Listing 6.

Listing 5 Example program code for pattern recognition by collecting.

```
(def squares xs
    (if (= xs nil) nil
                   (cons (* (head xs) (head xs))
                         (squares (tail xs)))))
(def withTwo xs
    (if (= xs nil) nil
                   (cons (+ (head xs) 2)
                         (withTwo (tail xs)))))
(def op xs
    (if (= xs nil) nil
                   (cons (- (* (head xs) 2) 2)
                         (op (tail xs)))))
(def positives xs
    (if (= xs nil) nil
                   (cons (>= (head xs) 0)
                         (positives (tail xs)))))
```

Listing 6 Pattern found in example code (variables marked with greek letters).

```
(def α xs (if (= xs nil) nil (cons β γ)))
```

5 Conclusion and Future Work

In this article, we have proposed a solution for automated introduction of new language abstractions based on patterns that in this study are understood as recurring structures in program code.

As part of the solution, two different approaches were experimentally developed to recognize language patterns: pattern recognition by comparing and by collecting. The first approach is based on comparing program fragments based on derivation trees of the applied Haskell grammar rules, generated by a complex set of analyzing tools [16]. Its principle lies in traversing particular derivation trees and recognizing the highest possible subtrees of the same structure.

While implementation of pattern recognition by comparing is relatively simple, and it is able to recognize most of the common recurring program structures, it does not allow substitution of particular subtrees by variables. That is, it cannot recognize some specific patterns. For instance, if we considered example code from Listing 5, the algorithm would only recognize the following pattern:

```
(def α xs (if (= xs nil) nil (cons (β (head xs) γ) (α (tail xs)))))
```

However, this covers only two of the four structurally similar program fragments. More general pattern (mentioned in Listing 6) was supported and recognized only by the second technique, which is pattern recognition by collecting. It reduced some drawbacks of the previous method, thus allowing substitution of subtrees by variables. On the other hand, the number of potential patterns inspected for each subtree needs to by limited since it is not possible to collect all of them for every size of a subtree.

Unlike in the previous experiment, the input of this algorithm consists of abstract syntax trees (not derivation trees) and its main principle lies in sequential generation of an associative array, with the keys of structural schemas and values of collected subtree lists corresponding to particular structural schemas.

The approach to pattern recognition by collecting is also more suitable for interactive use as a part of programming environment, because it allows incremental addition of program expressions into the pattern recognition process.

These experiments are significant to further part of the proposal focused on the concept of pattern based language enrichment using projectional editing. They show that it is possible to automatically find structural patterns in program code. To make more significant conclusions, it is necessary to perform experiments on greater set of programs and to further develop the algorithms. Development of the language enrichment environment is also needed to fully evaluate the proposal.

Further research in this area can be focused on the possibility to recognize patterns on higher level than the structure of code. These patterns may be scattered in the program code but semantically interconnected. Therefore, the pattern recognition process needs to have a high degree of knowledge about program semantics.

The contribution of the presented proposal for language enrichment is the new approach to the extension of programming language based on the needs of programmers [20]. It tries to combine the advantages of both linguistic and in-language abstractions, allowing language users to define new abstractions that are integrated into the language. In addition, the process of language enrichment is aided by automated patterns recognition.

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 [17], 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 [18], [13], as well as to perform a qualitative move from an automatic roundtrip engineering [3], [12] to the automated roundtrip software evolution.

References

1 Harold Abelson and Gerald J Sussman. Lisp: A language for stratified design. Technical report, Cambridge, MA, USA, 1987.

2 Harold Abelson and Gerald J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Electrical Engineering and Computer Science. The MIT Press, second edition, 1996.

3 Uwe Aßmann. Automatic roundtrip engineering. *Electronic Notes in Theoretical Computer Science*, 82(5):33–41, 2003.

4 D. Astapov. Using haskell with the support of business-critical information systems. *Practice of Functional Programming (in Russian)*, 2, 2009.

5 Frederick P. Brooks. No silver bullet: Essence and accidents of software engineering. *IEEE Computer*, 20(4):10–19, april 1987.

6 Sergey Dmitriev. Language oriented programming: The next programming paradigm. *JetBrains onBoard*, 1(2), November 2004. Available at `http://www.jetbrains.com/mps/docs/Language_Oriented_Programming.pdf`.

7 Sebastian Erdweg, Paolo G. Giarrusso, and Tillmann Rendel. Language composition untangled. In *Proceedings of Workshop on Language Descriptions, Tools and Applications (LDTA)*, 2012. to appear.

8 Martin Fowler. Language workbenches: The killer-app for domain specific languages? 2005. Available at `http://martinfowler.com/articles/languageWorkbench.html`.

9 Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

10 Paul Graham. *On Lisp.* Prentice Hall, 1994.

11 Lennart C. L. Kats, Eelco Visser, and Guido Wachsmuth. Pure and declarative syntax definition: Paradise lost and regained. In *Proceedings of Onward! 2010.* ACM, 2010.

12 Carsten Lohmann, Joel Greenyer, and Juanjuan Jiang. Applying triple graph grammars for pattern-based workflow model transformations. *Journal of Object Technology*, 6(9):253–273, 2007.

13 Ivan Luković, Pavle Mogin, Jelena Pavićević, and Sonja Ristić. An approach to developing complex database schemas using form types. *Software – Practice & Experience*, 37(15):1621–1656, December 2007.

14 Alex Ott. Using scheme in the development of "dozor-jet" family of products. *Practice of Functional Programming (in Russian)*, 2, 2009.

15 Simon Peyton Jones. Haskell 98 language and libraries – the revised report. Technical report, Cambridge England, 2003.

16 Emília Pietriková, Ľubomír Wassermann, Sergej Chodarev, and Ján Kollár. The effect of abstraction in programming languages. *Journal of Computer Science and Control Systems*, 4(1):137–142, 2011.

17 Jaroslav Porubän and Peter Václavík. Extensible language independent source code refactoring. In *AEI '2008: International Conference on Applied Electrical Engineering and Informatics*, pages 58–63, 2008.

18 Miroslav Sabo and Jaroslav Porubän. Preserving design patterns using source code annotations. *Journal of Computer Science and Control Systems*, 2(1):53–56, 2009.

19 Charles Simonyi, Magnus Christerson, and Shane Clifford. Intentional software. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 451–464, New York, NY, USA, 2006. ACM.

20 Guy L. Steele. Growing a language. *Higher-Order and Symbolic Computation*, 12:221–236, 1999.

21 Markus Voelter. *DSL Engineering: Designing, Implementing and Using Domain-Specific Languages.* CreateSpace Independent Publishing Platform, 2013.

22 Martin P. Ward. Language-oriented programming. *Software - Concepts and Tools*, 15(4):147–161, 1994.