ISBN 978-83-60810-22-4

Identification of Patterns through Haskell Programs Analysis

Ján Kollár, Sergej Chodarev, Emília Pietriková and Ľubomír Wassermann
Department of Computers and Informatics, Faculty of Electrical Engineering and Informatics
Technical University of Košice, Slovak Republic
E-mail: {jan.kollar, sergej.chodarev, emilia.pietrikova, lubomir.wassermann}@tuke.sk

Abstract—**Usage of appropriate high-level abstractions is very important for development of reliable and maintainable programs. Abstractions can be more effective if applied at the level of language syntax. To achieve this goal, analysis of programs based on the syntax is needed.**

This paper presents Haskell Syntax Analyzer tool that can be used for analysis of Haskell programs from the syntactic perspective. It allows to retrieve derivation trees of Haskell programs, visualize them and perform their statistical analysis. We also propose approach for recognition of recurring patterns in programs that can be used as a basis for automated introduction of abstractions into the language.

I. Introduction

ABSTRACTION is one of the fundamental concepts in computer science. Abstraction allows expressing things more simple by defining new more abstract concepts, that encapsulate complex expressions. This allows to hide implementation details.

For example, expression $\frac{-b+\sqrt{b^2-4ac}}{2a}$ for computing one root of quadratic equation $ax^2+bx+c=0$ can be simplified by introducing new abstract concept – *discriminant* (D). This form can be even more simplified by defining abstraction corresponding to the whole expression (see Fig. 1).

As a disadvantage of domain-specific languages (DSLs), the price of understanding DSL technology is often referred [1], [2]. This is caused by necessity of knowledge of the language design field and of the problem domain. In case of identification of software system field of where it is appropriate to deploy DSL, it is important to design the language suitably (syntax and notations). To increase the usability of a new DSL, the use of terminology and concepts of the target domain is essential.

The purpose of this work is not to derive a grammar from samples of different languages, but taking the full grammar of a language (in our case Haskell), it is evaluated.

II. Abstraction based on program patterns

To achieve the mentioned goals, first we need to solve the problem of recognition of recurring patterns in a code. Manual analysis of the code may be hard and tedious task. On the other hand, tools for automatic patterns recognition can greatly help in this task. Moreover, the recognition needs to be done at the level of program syntax.

Program patterns mean code fragments 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. After introduction of new abstraction based on a pattern, syntactic variables will become parameters of the abstraction.

Expressiveness of the language can be improved by the recognition of program patterns and introduction of abstractions based on them. Moreover, it allows more natural and strain-forward expression of programs.

This approach can be also useful for development of domain-specific dialects of programming languages. In order to implement this transition from general purpose language 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 points:

- focus on a particular domain,
- use of concepts from a domain,
- higher abstraction.

To achieve a connection with particular domain and a shift towards domain specificity, it is suitable to analyze existing programs (or program fragments) of 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.

The goal of this article is to propose a solution for automatized introduction of new language abstractions based on patterns found in a code.

III. Haskell Syntax Analyzer tool

To achieve the goal, *Haskell Syntax Analyzer* tool has been used. The aim of creating *Haskell Syntax Analyzer* tool is to gather needed information from Haskell programs to get a proper knowledge about used constructs in analyzed programs. As a result of the program analysis, derivation tree is produced, consisting of used rules of Haskell grammar [3]. Architecture of Haskell Syntax Analyzer tool consists of two parts – *generating infrastructure* and *analyzing infrastructure*.

The goal of generating infrastructure is to prepare tools that are used during the analysis of Haskell programs by the analyzing infrastructure.

The analyzing infrastructure contains lexer and parser of Haskell programs, intended for analysis of Haskell programs

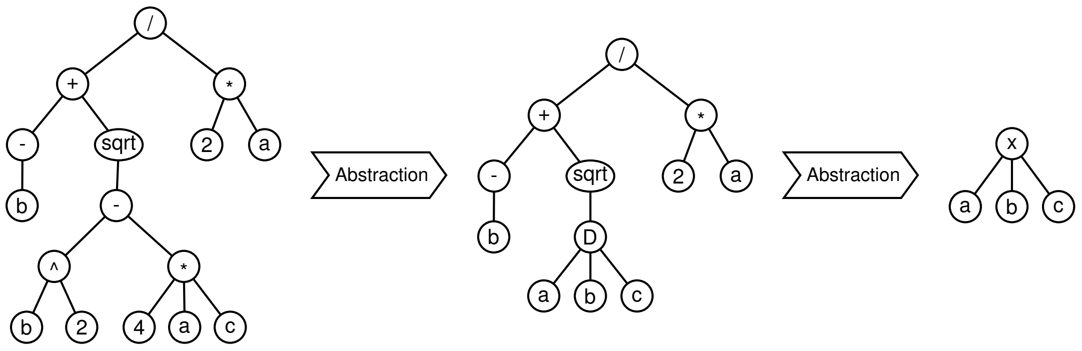

Fig. 1. Simplification of expression structure using abstraction.

into lexical units and then processing them into derivation trees.

Derivation trees are produced in the XML format, subsequently visualized using the Graphviz (Graph Visualization Software) tool [4] and further processed to retrieve statistical data on Haskell programs and to recognize common language patterns.

A. Haskell 98 Syntax

Haskell is a general purpose, purely functional programming language that provides higher-order functions, non-strict semantics, static polymorphic typing, user-defined algebraic data types, pattern-matching, list comprehensions, a module system, a monadic I/O system, and a rich set of primitive data types, including lists, arrays, arbitrary and fixed precision integers, and floating-point numbers [5].

In [6], Haskell syntax was processed into HTML format. This form was chosen as a basis for development of the parser. It uses extended Backus-Naur Form with the following additions:

- parentheses for grouping symbols,
- optional symbols marked using question mark (?),
- $0..n$ repetition marked with star ($\star$),
- $1..n$ repetition marked with plus (+).

To speed-up the development, it is necessary to use a parser generator tool. The choice of the appropriate parser generator depends on a class of the processed grammar. In Haskell grammar, it is possible to find several cases using the left recursion, e.g.:

```
aexp ::= qvar
     | gcon
     | literal
     | (exp)
     | (exp (,exp)+)
     | [exp (,exp)*]
     | [exp (,exp)?..(exp)?]
     | [exp | qual (,qual)*]
     | (exp_i qop)
     | (qop exp_i)
     | qcon  ( fbind (,fbind )* )?
     | aexp  fbind (,fbind )*
```

As the goal of the *Haskell Syntax Analyzer* tool is not to transform the language grammar but to process it in its original form, and because of the mentioned left recursion, Haskell grammar has been treated as the *LR* type.

In the grammar, reduce/reduce conflicts may be found, regarding several rules sharing the same right side. Thus, it is difficult for a parser to choose the right nonterminal to reduce. According to the above facts, it is appropriate to use GLR parser generator, as it is capable of parallel reduce of each nonterminal, trying to proceed with multiple possibilities.

In compliance with the project architecture, Haskell Grammar has been transformed into XML form in order to provide better representation of the grammar. The transparent form (XML) of the Haskell grammar is suitable for further processing, including generation of input for a parser generator.

As the parser generator, Bison [7] has been chosen, because of its ability to generate a GLR parser.

The Haskell language allows to use an indentation to define blocks of code. On the other hand, it still allows to define the blocks using braces and to separate the statements using semicolons. For this reason, it is required to introduce a separate step into lexical analysis, during which the layout defined by the white space characters is replaced by semicolons and braces.

Lexical analyzer is generated by the Flex tool [8] that is based on the specification produced according to the Haskell lexical grammar. After this step, the white space characters in the program are analyzed following the algorithm specified in Haskell 98 Report [3]. Based on this analysis, stream of tokens is extended by tokens corresponding to braces and semicolons.

B. Transformation of Haskell 98 Grammar

To be able to process the grammar programmatically, it is required to transform HTML form of grammar to more suitable representation. To transform the Haskell grammar to the appropriate form, a grammar transformer was created. This tool first transforms HTML representation of Haskell 98 grammar to XML representation that is more suitable for further processing.

XML grammar is then processed to create Java object model where XML elements are mapped to the instances of the grammar model classes. Java object model of Haskell grammar provides a better way of how to manipulate and operate on the Haskell grammar.

Mapping is shown in the example of a grammar rule:

```
gdrhs ::= gd = exp ( gdrhs )?
```

This rule is then transformed to the following XML fragment:

```
<nonterm id="n1" label="gdrhs">
  <sequence id="seq1">
    <nonterm id="n2" label="gd"/>
    <term id="equals" label="="/>
    <nonterm id="n3" label="exp"/>
    <option id="opt1">
      <nonterm id="n1" label="gdrhs"/>
    </option>
  </sequence>
</nonterm>
```

Java object model of Haskell grammar is then used to generate grammar specification in a format suitable for the Bison parser generator. During this process, grammar rules need to be transformed from EBNF form info BNF accepted by Bison. For this reason, new helper nonterminals are introduced. They correspond to constructs that can not be expressed directly in BNF, like repetition or optional expression. These nonterminals are specially marked according to their meaning, so it is possible to create derivation tree corresponding to original form of the grammar.

C. Grammar Ambiguity

Meanwhile the parsing, several ambiguities were detected. For example, rule `pat_i` of Haskell 98 grammar is defined as:

```
pat_i ::= pat_i ( qconop pat_i )?
     | - ( integer | float )
     | pat_10
```

After `pat_10` was reduced to `pat_i` (3rd alternative), it was possible to reduce `pat_i` to `pat_i` again (1st alternative). GLR parser generator, that is used within the Haskell Syntax Analyzer tool, disjoined at the mentioned point. As both ways led to the same nonterminal, they joined again. However, the parser was not capable to determine which way to use/throw away.

The problem was solved after modification of the critical rule:

```
pat_i ::= pat_10 ( qconop pat_i )?
```

A branch with '-' was moved to `pat_10`:

```
pat_10 ::= apat
     | gcon ( apat )+
     | - ( integer | float )
```

Another rule `exp_i` was changed by analogy of the previous rule. Original form:

```
exp_i a ::= exp_i ( qop exp_i )?
     | - exp_i
     | exp_10
```

After modification:

```
exp_i a ::= exp_10 ( qop exp_i )?
     | - exp_i
```

After such modifications, parser was able to process simple Haskell programs. Moreover, number of conflicts in the grammar were decreased.

D. Fixity Resolution

Another problem, that had been needed to be solved, was a resolution of fixity and precedence of operators. In Haskell 98 Report [3], operators precedence levels were defined using separate grammar rules (like exp_i for $0 \leq i \leq 9$). In the version of the grammar that had been used as a source for the transformation, the indexed rules were replaced by a single rule *exp_i*.

On the other hand, Haskell 2010 Report [9] defines expressions in a different way. It defines a single rule *infixexp* for all the precedence levels and associativities. The resolution of expressions is then performed after parsing. This approach is also appropriate for our purposes.

So after the parsing, resulting derivation trees are processed. All occurrences of the `exp_i` element are resolved using the algorithm described in the Haskell 2010 Report [9].

IV. Code Statistics

Using the developed tool, it was possible to compute some interested statistics based on a set of about 300 Haskell sample programs. Result of the analysis of a program is its derivation tree according to the language grammar. The derivation tree consists of terminal and nonterminal symbols in the grammar, 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 a 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 of a program and N represents a number of all symbols/nodes of the derivation tree.

Table I represents average occurrences greater than 0.01 of particular symbols in all programs of our sample. As it can be expected, variable names and expressions have the greatest frequency of all symbols.

It is possible to make such statistics for especially selected sample of programs for a specific domain. It will show which language elements are used in programs of the domain and which elements can be omitted from the domain-specific dialect.

Statistical analysis can also be used to partition a sample of programs into groups based on a usage of the language elements.

TABLE I
PROPORTION NUMBER OF SYMBOL OCCURRENCES

Symbol	Occurrence	Symbol	Occurrence
varid	0,093855	aexp	0,092660
fexp	0,092660	exp_10	0,063059
qvar	0,051154	exp_i	0,049428
exp	0,044523	var	0,037632
apat	0,033349	conid	0,026202
(	0,019259	)	0,019259
=	0,018341	decl	0,017526
;	0,017277	qop	0,016620
topdecl	0,016484	rhs	0,016164
pat_i	0,016099	pat_10	0,016099
qvarop	0,015110	gcon	0,014879
atype	0,013402	varsym	0,012450
qcon	0,011951	btype	0,011516
,	0,011309	funlhs	0,010876
type	0,010080		

V. PATTERNS RECOGNITION

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

```
$parents \leftarrow allParents(elements)$
$groups \leftarrow findGroups(parents)$
**if** $groups$ is empty **then**
  **return** $[groups]$
**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 with the following meaning:

- $allParents$ – returns a set of parents of all tree elements in a group;
- $findGroups$ – given a set of tree elements, returns list of groups of elements with similar subtrees;
- $mergeGroups$ – merges list of lists of groups 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.

Result of the algorithm is a list of groups of subtrees, where each group corresponds to a found pattern and contains all occurrences of the pattern.

VI. CONCLUSION

Set of tools called *Haskell Syntax Analyzer* that has been developed within this paper is intended to analyze programs based on the language syntax, resulting in providing appropriate derivation trees. In this paper, usage statistics of the Haskell syntactic symbols are provided, creating a vision of the language application within specific domains of use.

Moreover, the analysis made possible to accomplish pattern recognition in program codes, with the perspective of development of new language dialects, both general-purpose and domain-specific. The term of program patterns has been used for syntactically, and hence, also semantically equal program fragments occurring in a set of program samples.

The most significant contribution, that we expect based on the partial results presented in this paper, is the contribution for automated software evolution. Clearly, this would mean to shift from a language analyzer to the language abstracter, associating concepts to formal language constructs [10], [11], and formalizing them by means of these associations. In this way, we expect to integrate programming and modeling, associating the general purpose and domain-specific languages [12], [13], [14].

ACKNOWLEDGMENT

This work was supported by VEGA project No. 1/0015/10 “Principles and methods of semantic enrichment and adaptation of knowledge-based languages for automatic software development.”

REFERENCES

[1] M. Mernik, J. Heering, and A. M. Sloane, “When and how to develop domain-specific languages,” *ACM Comput. Surv.*, vol. 37, no. 4, pp. 316–344, 2005.
[2] M. Crepinsek, M. Mernik, B. Bryant, F. Javed, and A. Sprague, “Inferring context-free grammars for domain-specific languages,” *Electronic notes in theoretical computer science*, vol. 141, no. 4, pp. 99–116, 2005.
[3] S. Peyton Jones, *Haskell 98 Language and Libraries – The Revised Report*. Cambridge, England: Cambridge University Press, 2003.
[4] J. Ellson, E. Gansner, L. Koutsofios, S. North, and G. Woodhull, “Graphviz – open source graph drawing tools,” in *Graph Drawing*, ser. Lecture Notes in Computer Science, P. Mutzel, M. Jünger, and S. Leipert, Eds. Springer Berlin / Heidelberg, 2002, vol. 2265, pp. 594–597.
[5] B. O’Sullivan, J. Goerzen, and D. Stewart, *Real World Haskell*, 1st ed. O’Reilly Media, Inc., 2008.
[6] P. Hercek, “Haskell 98 report,” Available: http://www.hck.sk/users/peter/HaskellEx.htm, 2007.
[7] C. Donnelly and R. Stallman, *Bison: The Yacc-compatible Parser Generator*, 2010, available: http://www.gnu.org/software/bison/manual/.
[8] V. Paxson, W. Estes, and J. Millaway, *Lexical Analysis With Flex*, 2007, available: http://flex.sourceforge.net/manual/.
[9] S. Marlow, “The Haskell 2010 Language Report,” Available: http://www.haskell.org/onlinereport/haskell2010/, 2010.
[10] J. Porubän and P. Václavík, “Extensible language independent source code refactoring,” in *AEI ’2008 : International Conference on Applied Electrical Engineering and Informatics, Greece, Athens, September 8-11*. Košice: FEI TU, 2008, pp. 58–63.
[11] J. Porubän and M. Sabo, “Jessine: Integrating rules in enterprise software applications,” *Journal of Information, Control and Management Systems*, vol. 7, no. 1, pp. 81–88, 2009.
[12] M. Sabo and J. Porubän, “Preserving design patterns using source code annotations,” *Journal of Computer Science and Control Systems*, vol. 2, no. 1, pp. 53–56, 2009.
[13] P. Václavík, “Application domain name-based analysis,” *Journal of Computer Science and Control Systems*, vol. 2, no. 2, pp. 66–69, 2009.
[14] I. Luković, P. Mogin, J. Pavićević, and S. Ristić, “An approach to developing complex database schemas using form types,” *Software Practice & Experience*, vol. 37, no. 15, pp. 1621–1656, 2007.