Informatics and Computer Languages Laboratory
(Adaptive Languages and Software Systems)
http://icl.kpi.fei.tuke.sk/

Software Evolution based on Software Language Engineering

Ján Kollár, Jaroslav Porubän, Peter Václavík, Michal Forgáč, Miroslav Sabo, Žubomír Wassermann, František Mrázik, Michal Vagač, and Peter Klobušník

Jan.Kollar@tuke.sk
Introduction

1. Languages are of the utmost importance in the world of software.

2. Software language engineering – a new field of computer science, promising the evolution of software, instead of the development.

3. While essential complexity is inherent in the problem being solved, accidental complexity is an artifact of the solution, i.e., the software written to solve the problem.

4. The only way to overcome or avoid the negative effects of software aging is by placing change and evolution in the center of software development process.

5. In software engineering, the important role of abstraction is understood superficially as it is in fact identified with insufficiency.

6. In software language engineering (SLE), abstraction is a simple but powerful tool for making things in more general manner.
Principles of Software Language Engineering 1-3

• Language is a tool communicating with both software and people.

• Language is a tool for observing the properties of software and guarding the decisions for change.

• Language is an open system, changed using metalanguage, not directly.

• Formal languages in SLE are classified as that being able to express some properties using computer. It means that machine languages, programming languages, modeling languages, specification languages and even subsets of natural languages are all formal languages.
Principles of Software Language Engineering 2-3

- Software is observed and affected by language, not directly.
- Language is mutated by changes of informal notion as well as by changes of software properties.
- There is no need for narrowing semantical gap between specification and implementation, since it does not exist. By other words, horizontal semantical insufficiency is not acceptable.
- However, semantical insufficiency in vertical direction is acceptable, and even useful, since it yields semantical convergence and, as a result, stability of software accidental complexity.
• There is no strict boundary between abstract and concrete syntax – the language is simply the subject of change.

• Software evolution is quite different from software development; evolution is the matter of language transformation, with automated software transformation, while development focuses on software transformation directly.

• Language symmetry and semantical equivalence between informal notion and implementation are the essential characteristics during mutation. The form of language is irrelevant.

• Minimum accidental complexity of software is the essential criterion in each mutation step.
Basically, SLE focuses on the following challenge.

*How to mutate highly reflective language by metalanguage to achieve change of software without affecting software manually.*
SLE Conception: Mutation Step
Selected Properties of Mutable Language

• The necessity for aligning language to problem domain, introducing catastrophic scenario of rapid growth of accidental complexity, if we ignore this requirement.

• Need for language symmetry alignment, otherwise language may become not understandable.
Aligning Language to Problem Domain

Problem – in functional problem area:

This is text
in three
subsequent lines.

should be processed, as follows:

> lines "This is text
in three
subsequent lines."
["This is text","in three","subsequent lines."]
> lines "This is text
in three
subsequent lines."
["This is text","in three","subsequent lines."]
Solution

lines :: String -> [String]
lines = foldr breakon_nl []
  where x 'breakon_nl' xss | x == '\n' = [[]] ++ xss
                          | otherwise = no_nl xss
  where no_nl [] = [[x]]
        no_nl (xs:xss) = [[x] ++ xs] ++ xss

Total of 180 characters excluding spaces. We have used functional programming language in functional programming domain.
Shift to Synthesis Problem Area

> unlines ["This is text","in three","subsequent lines."]
"This is text
in three
subsequent lines.
"

unlines :: [String] -> String
unlines = foldr insert_nl []
    where xs ‘insert_nl‘ ys = xs ++ [’\n’] ++ ys

lines (unlines xss) = xss
lines [x] = lines ([x] ++ [’\n’]) = [[x]]

Total of 3684 characters. The result is the same, but solving problem in different problem area, accidental complexity has increased more than 20 times.
Abstraction in SLE

breakOn :: Char -> String -> [String]

such that breakOn ‘\n’ is equivalent to lines, breakOn ‘ ’ separates words of one line, and in general, breakOn c separates all substrings of a string on character c.
Language Symmetry

> countFile
File name: D1.txt
File D1.txt has 180 characters except spaces.

countFile = putStrLn "File name: " >>=
  _ -> getLine >>= \
x -> readFile x >>=
  (return . length . filter (/= ' ')) >>=
  z -> putStrLn ("File " ++ x ++ " has "
  ++ show z ++ " characters except spaces."")

Since of target-level lambda abstractions occurence in this program, it is hardly understandable, so it is evident that Haskell is asymmetric with respect to imperative paradigm.
## More Symmetric Interface

Table user interface, the symmetry is renewed – values submitted to monad are clearly used in next actions and/or transformations, and program is more transparent.

<table>
<thead>
<tr>
<th>Step</th>
<th>Monad values</th>
<th>Monad Action or Transformation</th>
<th>Value submitted</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td></td>
<td><code>putStr &quot;File name:&quot;</code></td>
<td>()</td>
</tr>
<tr>
<td>2.</td>
<td>()</td>
<td><code>getline</code></td>
<td>x</td>
</tr>
<tr>
<td>3.</td>
<td>x</td>
<td><code>readFile x</code></td>
<td>●</td>
</tr>
<tr>
<td>4.</td>
<td>●</td>
<td><code>(return.length.filter (/= ' '))</code></td>
<td>z</td>
</tr>
<tr>
<td>5.</td>
<td>z</td>
<td><code>putStr (&quot;File &quot;++x++&quot; has &quot;++show z ++&quot; characters except spaces.&quot;)</code></td>
<td>()</td>
</tr>
</tbody>
</table>
Projects

- [2001-2003] Binding the process functional language to MPI.
- [2009-2011] Composition and Evolution of Software Languages
- [2009-2010] Aspect Monitoring and Reconfiguration of Complex Software Systems
Further Research

- Observation and adaptation rules
- Meta-level language transformations
- Rules driven abstraction and composition of languages
- Mutation of grammars and semantics
- Design efforts for domain-adaptive language
- Implementation efforts for domain-adaptive language
Conclusion

- SLE integrates specification, programming, modeling and runtime requirements.
- Methodology of SLE is scientific, but goal is practical.
- SLE makes software production highly efficient in application areas, such as embedded systems, user interfaces, data mining, knowledge based systems, advisory systems, reactive systems, agile systems, etc.
- The world of computer networking and distributed processing requires flexible change and adaptation, reducing subsequently accidental complexity of software.
- SLE brings programmers and designers to the position of observers, decision supervisors, and innovative solution founders.
- Strong demand for SLE approaches in the near future by software production companies is expected.
Thank you for your attention