CS421 - Summer 2024 - Project Report

Zhentao Xu (zx46@illinois.edu)

Submission

Project Code: https://github.com/zx46uiuc/zx46-cs421-project
Project Report: this document.

Paper Link:

Software Metrics: Measuring Haskell

Overview

The paper discusses software metrics tailored for Haskell, focusing on improving code quality and identifying areas for testing and refactoring. Unlike imperative languages, functional languages like Haskell have been less explored for software metrics. The metrics are essential tools for managing complexity and ensuring the quality of large-scale systems by offering objective, validated measures of various code attributes. I am interested in this paper because the discussed metrics and methodologies are invaluable for systematic and efficient software maintenance and improvement. This is especially true for large-scale software in the industrial, where code complexity estimation is essential in guaranteeing code stability in production.

In this project, I present a summary of key learnings regarding metrics in the Haskell programming language. Specifically, I implemented a prototype of several critical metrics that can be integrated into a Haskell program for measurement (see my repository link here) and I describe the implementation details. I also test and validate the code by applying the metrics on two mock AST code with various complexities and see how the metrics can correctly identify the complexity from aspects of various metric perspectives.

Paper Summary

The paper authors introduce metrics for evaluating the complexity of Haskell programs, focusing on attributes such as patterns, distance, callgraph, and function characteristics. Key metrics include Pattern Size (PSIZ), Number of Pattern Variables (NPVS), and Pathcount (PATH). These metrics help identify complex areas in code, aiding in better testing and maintenance. The paper also emphasizes the need for further development of these metrics for enhanced accuracy and applicability.

Metrics for Measuring Haskell Program

- Pattern: patterns are widely used in Haskell programs thus it's interesting to understand how they affect the complexity of a program. The metrics proposed in this paper includes
    - **Pattern Size (PSIZ)**: Counts the number of components in the pattern's abstract syntax tree, assuming larger patterns are more complex.
    - **Number of Pattern Variables (NPVS)**: Measures the identifiers introduced by patterns, affecting the cognitive load on programmers.
    - **Number of Overridden/Overriding Pattern Variables**: Considers the impact of variable overriding on potential program errors.
    - **Number of Constructors (PATC)**: Tracks the use of data type constructors, which can increase the complexity of understanding the program.
    - **Number of Wildcards (WILD):** Wildcards, while often ignored, indicate the structure of patterns and their complexity.
    - **Depth of Nesting**: Measures the complexity added by nested patterns, using metrics like the sum of the depth of nesting (SPDP) and maximum depth of nesting.
- Distance Metrics: The paper also discusses metrics related to the "distance" between declarations and their usage in the code, which can impact the likelihood of errors:
    - **Number of New Scopes**: Measures the distance by counting new scopes between a declaration and its use.
    - **Number of Declarations Brought into Scope**: Extends the previous metric by counting declarations introduced in new scopes.
    - **Number of Source Lines**: A spatial measure, counting the number of lines between a declaration and its use.
    - **Number of Parse Tree Nodes**: Counts nodes in the parse tree to measure the amount of code between two points, providing a consistent measure of distance.
    - **Cross-Module Distance**: Specifically measures the distance when dealing with modules, considering the distance from the use of an identifier to the import statement and the start of the module.
- Callgraph Attributes: The paper introduces several metrics related to calligraphy, which represent function calls in Haskell programs. These metrics help in understanding the complexity and dependencies within the code:
    - **Strongly Connected Component Size (SCCS)**: Measures the size of cyclic subgraphs in a callgraph. Larger SCCS values suggest greater interdependence among functions, potentially increasing maintenance complexity and the ripple effect of changes.
    - **Indegree (IDEG):** Counts the number of functions that call a particular function, indicating its reuse. Functions with high IDEG are crucial, as changes in them can affect many other parts of the program.
    - **Outdegree (OUTD):** Indicates the number of functions called by a particular function. Higher OUTD suggests more dependencies, increasing the likelihood of changes when any dependent function is modified.

- **Arc-to-Node Ratio (ATNR):** Measures the density of connections in the callgraph, indicating how "busy" a function is. A higher ATNR implies greater interaction complexity.
- **Callgraph Depth (CGDP) and Callgraph Width (CGWD)**: These metrics analyze the size of the call graph subgraph for a function, where depth indicates the longest path and width indicates the breadth of the callgraph. These metrics help in assessing the comprehensibility and potential error-proneness of the code.

* Function Attributes:
    - **Pathcount (PATH)**: Measures the number of logical execution paths through a function. It's a predictor of faults, as a higher number of paths often correlates with higher complexity and potential errors. The paper illustrates hidden paths that can arise in pattern matching and guard usage in Haskell, which might not be immediately obvious.
    - **Operators (OPRT)** and Operands (OPRD): These metrics, adapted from Halstead's metrics, measure the size of a function by counting operators and operands. Larger functions, indicated by higher counts, are often more complex and harder to maintain. This pair of metrics helps in assessing the complexity and potential error-proneness of functions.

Validation

The authors also validate their proposed metrics by analyzing two case study programs. They measure the metrics across the development lifecycle and correlate them with the number of bug-fixing changes. The aim is to identify metrics that correlate well with bug fixes, indicating areas in the code that may benefit from more rigorous testing of refactoring. The study highlights the potential of using these metrics to target software engineering efforts effectively.

Implementation

Overview

The implementation includes individual functions for each metric, where each function takes an expression as input and outputs the corresponding metric value. Haskell's pattern matching is used to adapt the calculations based on the expression type in the AST. The metrics were validated using simple and complex Haskell code examples, ensuring that more complex ASTs yield higher metric values. (Please checkout my repository for details)

Explanation of Implementation Details

- **PSIZ Implementation:** Calculates the pattern size by examining each expression type, including base cases like "Var" and complex cases like "App" (function application). My implementation is copy-pasted here for reference and demonstrating the method.

```
patternSize :: Expr -> Int
patternSize (Var _) = 1
patternSize (Constr _ exprs) = 1 + sum (map patternSize exprs) -- one for
this expression, and the summation of children expression size., recursively
patternSize (App e1 e2) = patternSize e1 + patternSize e2 -- recursion.
patternSize (Lam _ e) = patternSize e
patternSize (Let bindings e) =
 sum (map (patternSize . snd) bindings) + patternSize e
```

- NPVS Implementation: The numPatternVars function calculates the number of pattern variables in an expression by recursively traversing the AST. It counts each variable in a pattern and sums up the counts for nested expressions, including variables in lambda abstractions and let bindings.
- PATC Implementation: The numConstructors function counts the number of constructors used in an expression. It treats each constructor occurrence as one unit and aggregates the counts for nested constructors within the AST.
- PATH Implementation: The pathCount function determines the number of logical execution paths through a given expression. It counts each path for variable and constructor expressions, multiplying paths for applications to account for branching, and sums paths in let bindings and lambda abstractions.
- DON Implementation: The depthOfNesting function calculates the depth of nesting in an expression by determining the maximum depth of nested expressions. It handles various expression types, including variables, constructors, applications, and let bindings, incrementing the depth count accordingly.

Status of Project

The metrics work correctly and align with the expectation that more complex code will have corresponding higher metric values. Future work could involve integrating additional metrics from other sources to extend the calculator's capabilities.

Tests

I created two example of Haskell code and corresponding AST for them, and feed the AST into my metrics calculator for the several metrics mentioned in this paper. The number is correct.

<table>
<tr><th>Metrics</th><th>Example 1 (Simple Example)</th><th>Example 2 (Complex Example)</th></tr>
<tr><td>Haskell Code</td><td><pre><code>data Tree = Leaf Int | Node Tree
Tree deriving (Show)

sumTree :: Tree -> Int
sumTree (Leaf n) = n
sumTree (Node left right) = sumTree
left + sumTree right

doubleTree :: Tree -> Tree
doubleTree (Leaf n) = Leaf (2 * n)
doubleTree (Node left right) = Node
(doubleTree left) (doubleTree
right)

main :: IO ()
main = do
 let tree = Node (Leaf 1) (Node
(Leaf 2) (Leaf 3))
 print (sumTree tree)
 print (doubleTree tree)
</code></pre></td><td><pre><code>data Tree = Leaf Int | Node Tree Tree
deriving (Show)

sumTree :: Tree -> Int
sumTree (Leaf n) = n
sumTree (Node left right) = sumTree left +
sumTree right

doubleTree :: Tree -> Tree
doubleTree (Leaf n) = Leaf (2 * n)
doubleTree (Node left right) = Node
(doubleTree left) (doubleTree right)

applyAndPrintTree :: (Tree -> Tree) -> Tree
-> IO ()
applyAndPrintTree f tree = do
 let caseResult = f tree
 let newTree = Node caseResult (Leaf 0)
 printTree newTree

printTree :: Tree -> IO ()
printTree (Leaf n) = print n
printTree (Node left right) = do
 printTree left
 printTree right

main :: IO ()
main = do
 let tree = Node (Leaf 1) (Node (Leaf 2)
(Node (Leaf 3) (Leaf 4)))
 print (sumTree tree)
 print (doubleTree tree)
 applyAndPrintTree doubleTree tree
</code></pre></td></tr>
<tr><td>AST</td><td><pre><code>demoASTSimple :: [Expr]
demoASTSimple =
 [Constr
     "Node"
     [Constr "Leaf" [Var "1"]
</code></pre></td><td><pre><code>demoASTComplex :: [Expr]
demoASTComplex =
 [Constr
     "Node"
     [Constr "Leaf" [Var "1"]
</code></pre></td></tr>
</table>

<table>
<tr><td></td><td><code>, Constr "Node" [Constr "Leaf"</code>
<code>[Var "2"], Constr "Leaf" [Var "3"]]</code>
<code>]</code>
<code>, Lam ["tree"] (App (Var</code>
<code>"sumTree") (Var "tree"))</code>
<code>, Lam ["tree"] (App (Var</code>
<code>"doubleTree") (Var "tree"))</code>
<code>]</code></td><td><code>, Constr "Node"</code>
<code>[Constr "Leaf" [Var "2"]</code>
<code>, Constr "Node"</code>
<code>[Constr "Leaf" [Var "3"]</code>
<code>, Constr "Leaf" [Var "4"]</code>
<code>]</code>
<code>]</code>
<code>]</code>
<code>, Lam ["tree"] (App (Var "sumTree") (Var</code>
<code>"tree"))</code>
<code>, Lam ["tree"] (App (Var "doubleTree") (Var</code>
<code>"tree"))</code>
<code>, Lam ["f", "tree"]</code>
<code>(Let</code>
<code>[("caseResult", App (Var "f") (Var</code>
<code>"tree"))</code>
<code>, ("newTree", Constr "Node" [Var</code>
<code>"caseResult", Constr "Leaf" [Var "0"]])</code>
<code>]</code>
<code>(App (Var "printTree") (Var</code>
<code>"newTree"))</code>
<code>)</code>
<code>]</code></td></tr>
<tr><td>Pattern size</td><td>12</td><td>23</td></tr>
<tr><td>Number of pattern variables</td><td>9</td><td>18</td></tr>
<tr><td>Number of constructors</td><td>5</td><td>9</td></tr>
<tr><td>Pathcount</td><td>10</td><td>19</td></tr>
<tr><td>Degree of Nesting</td><td>10</td><td>16</td></tr>
<tr><td>Number of new scopes</td><td>2</td><td>4</td></tr>
<tr><td>Number of declarations</td><td>2</td><td>6</td></tr>
<tr><td>Number of source lines</td><td>16</td><td>33</td></tr>
<tr><td>Number of parse tree nodes</td><td>16</td><td>33</td></tr>
</table>

Listing

```haskell
-- AST representation (simplified for demonstration purposes)
data Expr
 = Var String -- A variable represented by a string
 | Constr String [Expr] -- A constructor with a name and a list of expressions
 | App Expr Expr -- Application of one expression to another
 | Lam [String] Expr -- Lambda abstraction with a list of bound variables and an expression
 | Let [(String, Expr)] Expr -- Let bindings with a list of variable-expression pairs and a body expression
 deriving (Show, Eq)

-- Example AST for a simple demo Haskell program
demoASTSimple :: [Expr]
demoASTSimple =
 [ Constr
     "Node"
     [ Constr "Leaf" [Var "1"]
     , Constr "Node" [Constr "Leaf" [Var "2"], Constr "Leaf" [Var "3"]]
     ]
 , Lam ["tree"] (App (Var "sumTree") (Var "tree"))
 , Lam ["tree"] (App (Var "doubleTree") (Var "tree"))
 ]

-- Example AST for a more complex demo Haskell program
demoASTComplex :: [Expr]
demoASTComplex =
 [ Constr
     "Node"
     [ Constr "Leaf" [Var "1"]
     , Constr
         "Node"
         [ Constr "Leaf" [Var "2"]
         , Constr "Node" [Constr "Leaf" [Var "3"], Constr "Leaf" [Var "4"]]
         ]
     ]
 , Lam ["tree"] (App (Var "sumTree") (Var "tree"))
 , Lam ["tree"] (App (Var "doubleTree") (Var "tree"))
```

```haskell
 , Lam
     ["f", "tree"]
     (Let
        [ ("caseResult", App (Var "f") (Var "tree"))
        , ( "newTree"
          , Constr "Node" [Var "caseResult", Constr "Leaf" [Var "0"]])
        ]
        (App (Var "printTree") (Var "newTree")))
 ]

-- Current demo AST used for testing
demoAST :: [Expr]
demoAST = demoASTComplex

-- Metric 1: Pattern Size (PSIZ)
patternSize :: Expr -> Int
patternSize (Var _) = 1
patternSize (Constr _ exprs) = 1 + sum (map patternSize exprs)
patternSize (App e1 e2) = patternSize e1 + patternSize e2
patternSize (Lam _ e) = patternSize e
patternSize (Let bindings e) =
 sum (map (patternSize . snd) bindings) + patternSize e

-- Metric 2: Number of Pattern Variables (NPVS)
numPatternVars :: Expr -> Int
numPatternVars (Var _) = 1 -- variable is contributing to variable.
numPatternVars (Constr _ exprs) = sum (map numPatternVars exprs)
numPatternVars (App e1 e2) = numPatternVars e1 + numPatternVars e2
numPatternVars (Lam vars e) = length vars + numPatternVars e
numPatternVars (Let bindings e) =
 sum (map (numPatternVars . snd) bindings) + numPatternVars e

-- Metric 3: Number of Constructors (PATC)
numConstructors :: Expr -> Int
numConstructors (Var _) = 0
numConstructors (Constr _ exprs) = 1 + sum (map numConstructors exprs) -- constructor contribute to 1
numConstructors (App e1 e2) = numConstructors e1 + numConstructors e2
numConstructors (Lam _ e) = numConstructors e
numConstructors (Let bindings e) =
 sum (map (numConstructors . snd) bindings) + numConstructors e
```

```haskell
-- Metric 4: Depth of Nesting (DON)
depthOfNesting :: Expr -> Int
depthOfNesting (Var _) = 1
depthOfNesting (Constr _ exprs) = 1 + maximum (map depthOfNesting exprs)
depthOfNesting (App e1 e2) = 1 + max (depthOfNesting e1) (depthOfNesting e2)
depthOfNesting (Lam _ e) = 1 + depthOfNesting e
depthOfNesting (Let bindings e) =
 1 + max (maximum (map (depthOfNesting . snd) bindings)) (depthOfNesting e)

--- Function Call
-- Metric 4: Pathcount (PATH)
pathCount :: Expr -> Int
pathCount (Var _)          = 1
pathCount (Constr _ exprs) = 1 + sum (map pathCount exprs)
pathCount (App e1 e2)      = pathCount e1 * pathCount e2 -- all possible paths between e1 and e2.
pathCount (Lam _ e)        = pathCount e
pathCount (Let bindings e) = sum (map (pathCount . snd) bindings) + pathCount e

-- Distance Metrics
-- Distance Metric 1: Number of New Scopes
numNewScopes :: Expr -> Int
numNewScopes (Var _) = 0
numNewScopes (Constr _ exprs) = sum (map numNewScopes exprs)
numNewScopes (App e1 e2) = numNewScopes e1 + numNewScopes e2
numNewScopes (Lam _ e) = 1 + numNewScopes e
numNewScopes (Let bindings e) =
 1 + sum (map (numNewScopes . snd) bindings) + numNewScopes e

-- Distance Metric 2: Number of Declarations Brought into Scope
numDeclarationsInScope :: Expr -> Int
numDeclarationsInScope (Var _) = 0
numDeclarationsInScope (Constr _ exprs) = sum (map numDeclarationsInScope exprs)
numDeclarationsInScope (App e1 e2) =
 numDeclarationsInScope e1 + numDeclarationsInScope e2
numDeclarationsInScope (Lam vars e) = length vars + numDeclarationsInScope e
numDeclarationsInScope (Let bindings e) =
 length bindings
   + sum (map (numDeclarationsInScope . snd) bindings)
   + numDeclarationsInScope e

-- Distance Metric 3: Number of Source Lines
```

```haskell
numSourceLines :: Expr -> Int
numSourceLines (Var _) = 1
numSourceLines (Constr _ exprs) = 1 + sum (map numSourceLines exprs)
numSourceLines (App e1 e2) = 1 + numSourceLines e1 + numSourceLines e2
numSourceLines (Lam _ e) = 1 + numSourceLines e
numSourceLines (Let bindings e) =
 1
   + length bindings
   + sum (map (numSourceLines . snd) bindings)
   + numSourceLines e

-- Distance Metric 4: Number of Parse Tree Nodes
numParseTreeNodes :: Expr -> Int
numParseTreeNodes (Var _) = 1
numParseTreeNodes (Constr _ exprs) = 1 + sum (map numParseTreeNodes exprs)
numParseTreeNodes (App e1 e2) = 1 + numParseTreeNodes e1 + numParseTreeNodes e2
numParseTreeNodes (Lam _ e) = 1 + numParseTreeNodes e
numParseTreeNodes (Let bindings e) =
 1
   + length bindings
   + sum (map (numParseTreeNodes . snd) bindings)
   + numParseTreeNodes e

main :: IO ()
main = do
 putStrLn
   $ "Pattern size (PSIZ) of the demo AST: "
       ++ show (sum (map patternSize demoAST))
 putStrLn
   $ "Number of pattern variables (NPVS) of the demo AST: "
       ++ show (sum (map numPatternVars demoAST))
 putStrLn
   $ "Number of constructors (PATC) of the demo AST: "
       ++ show (sum (map numConstructors demoAST))
 putStrLn
   $ "Pathcount (PATH) of the demo AST: " ++ show (sum (map pathCount demoAST))
 putStrLn
   $ "Depth of nesting (DON) of the demo AST: "
       ++ show (sum (map depthOfNesting demoAST))
 putStrLn
   $ "Number of new scopes in demoAST: "
```

```haskell
      ++ show (sum (map numNewScopes demoAST))
putStrLn
  $ "Number of declarations brought into scope in demoAST: "
      ++ show (sum (map numDeclarationsInScope demoAST))
putStrLn
  $ "Number of source lines in demoAST: "
      ++ show (sum (map numSourceLines demoAST))
putStrLn
  $ "Number of parse tree nodes in demoAST: "
      ++ show (sum (map numParseTreeNodes demoAST))
```

How to Run Code

```
stack ghci ./main.hs


main
```

Reference

[1] Ryder, Chris, and Simon Thompson. "Software metrics: measuring haskell." (2005).