Reverse Engineering of Complex Software Systems via Static Analysis

Melinda Tóth, István Bozó, Zoltán Horváth

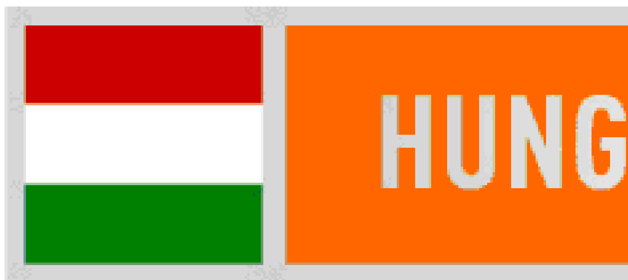
Created by XMLmind XSL-FO Converter.

Reverse Engineering of Complex Software Systems via Static Analysis

Melinda Tóth, István Bozó, Zoltán Horváth

Publication date 2014
Copyright © 2014 Melinda Tóth, István Bozó, Zoltán Horváth

Supported by TÁMOP-4.1.2.A/1-11/1-2011-0052.



The proj
the Euro
by the E

Table of Contents

1. Lecture 1 ... 1
1. Syllabus ... 1
1.1. Syllabus ... 1
2. Static Analysis ... 1
2.1. Static Analysis ... 1
2.2. Reverse Engineering of Software ... 1
3. Introduction to Erlang ... 1
3.1. Functional Programming ... 1
3.2. Properties ... 2
3.3. History ... 2
3.4. Erlang – Properties ... 2
3.5. Erlang – Ericsson Language ... 3
3.6. When To Use Erlang? ... 3
3.7. Who Uses Erlang? ... 3
2. Lecture 2 ... 5
1. The Syntax of Erlang programs ... 5
1.1. Language elements ... 5
1.2. Language elements – Examples ... 5
1.3. Constants and Variables ... 5
1.4. Constants and Variables – Examples ... 5
1.5. Functions ... 6
1.6. Functions – Example ... 6
1.7. Patterns ... 6
1.8. Patterns – Example ... 6
1.9. Expressions 1. ... 7
1.10. Expressions – List ... 7
1.11. Expressions 2. ... 7
1.12. Expressions – Branching ... 8
1.13. Expressions 3. ... 8
1.14. Expressions – Branching ... 8
1.15. Expressions 4. ... 9
1.16. Expressions – Branching ... 9
1.17. Expressions 5. ... 10
1.18. Expressions – Funexpressions ... 10
1.19. Expressions 6. ... 10
1.20. Expressions – Records ... 10
1.21. Expressions 7. ... 11
1.22. Expressions – Binary ... 11
1.23. Guards ... 11
3. Lecture 3 ... 12
1. Abstract syntax tree ... 12
1.1. Abstract syntax tree (AST) ... 12
1.2. Abstract syntax tree (AST) ... 12
1.3. About RefactorErl ... 12
1.4. AST in the RefactorErl ... 12
1.5. Parser in RefactorErl ... 12
1.6. Rule syntax ... 12
1.7. Module attribute ... 13
1.8. Module attribute – Example ... 13
1.9. Export attribute ... 13
1.10. Export attribute – Example ... 13
1.11. Import attribute ... 13
1.12. Record definition ... 13
1.13. Record definition – Example ... 14
1.14. Function ... 14
1.15. Function Clause – Example ... 14

Created by XMLmind XSL-FO Converter.

1.16. Case expression ... 14
1.17. Case expression – Example ... 15
1.18. If expression ... 15
1.19. Receive expression ... 15
2. Symbol table ... 15
2.1. Symbol table ... 15
4. Lecture 4 ... 16
1. Preprocessing ... 16
1.1. Preprocessor ... 16
2. Semantic Program Graph ... 16
2.1. Semantic graph model ... 16
2.2. Semantic graph ... 16
2.3. Mathematical model ... 16
2.4. Graph Schema ... 17
2.5. Graph corresponds to the given Schema ... 17
2.6. Graph traversal (`path expression`) ... 17
2.7. Graph traversal (`path expression` evaluation) ... 17
2.8. Graph traversal (filtering the result) ... 17
2.9. Graph traversal (additional functions) ... 18
2.10. Examples ... 18
5. Lecture 5 ... 19
1. Introduction ... 19
1.1. Architecture of RefactorErl ... 19
2. Lexical layer ... 19
2.1. Lexical Schema ... 19
2.2. Lexical information ... 19
2.3. Token information ... 19
3. Syntactic layer ... 19
3.1. Syntactic Schema ... 20
3.2. Syntactic Schema ... 20
3.3. File information ... 20
3.4. Form information ... 20
3.5. Clause information ... 20
3.6. Expression information ... 21
3.7. Type expression information ... 21
4. Semantic layer ... 21
4.1. Semantic Schema ... 21
4.2. Semantic Schema ... 21
4.3. Semantic Schema ... 22
4.4. Module information ... 22
4.5. Function information ... 22
4.6. Function information ... 23
4.7. Variable information ... 23
4.8. Context information ... 23
4.9. Record information ... 23
4.10. Record field information ... 24
4.11. ETS table information ... 24
4.12. PID information ... 24
4.13. Environment information ... 24
6. Lecture 6 ... 25
1. Data-flow graph ... 25
1.1. Data-flow ... 25
1.2. Reaching definition analysis ... 25
1.3. Data-Flow analysis in RefactorErl ... 25
1.4. Kinds of Data-Flow edges ... 25
1.5. Kinds of Data-Flow edges – Examples ... 26
1.6. Notations used in the formal rules ... 26
1.7. Data-flow rule: Variable ... 26
1.8. Variable – Example ... 26
1.9. Data-flow rule: Match expression ... 27

Created by XMLmind XSL-FO Converter.

1.10. Match Expression – Example ... 27
1.11. Data-flow rule: Pattern ... 27
1.12. Data-flow rule: Unary operator ... 27
1.13. Data-flow rule: Infix operator ... 27
1.14. Infix operator – Example ... 27
1.15. Data-flow rule: Parenthesis ... 27
1.16. Data-flow rule: Tuple expression ... 27
1.17. Tuple expression – Example ... 28
1.18. Data-flow rule: Tuple pattern ... 28
1.19. Tuple pattern – Example ... 28
1.20. Data-flow rule: List expression ... 28
1.21. Data-flow rule: List comprehension ... 28
1.22. Data-flow rule: List pattern ... 28
1.23. Data-flow rule: BIF 1 ... 28
1.24. BIF 1 – Example ... 28
1.25. Data-flow rule: BIF 2 ... 29
1.26. Data-flow rule: BIF 3 ... 29
1.27. Data-flow rule: Case expression ... 29
1.28. Case expression – Example ... 29
1.29. Data-flow rule: If expression ... 29
1.30. Data-flow rule: Function call ... 30
1.31. Function call – Example ... 30
1.32. Data-flow rule: Function call 2 ... 31
1.33. Data-flow rule: Try expression ... 31
1.34. Data-flow rule: Catch expression ... 31
1.35. Data-flow rule: Block expression ... 32
1.36. Data-flow rule: Send and receive expression ... 32
1.37. Data-flow rule: Fun expression 1 ... 32
1.38. Data-flow rule: Fun expression 2 ... 33
1.39. Data-flow rule: Fun expression 3 ... 33
1.40. Data-flow rule: Dynamic function call 1 ... 34
1.41. Data-flow rule: Dynamic function call 2 ... 34
1.42. Data-flow rule: Dynamic function call 3 ... 35
1.43. Data-flow rule: Dynamic function call 4 ... 35
7. Lecture 7 ... 36
1. Data-flow graph ... 36
1.1. Data-Flow reaching ... 36
2. 0th order data-flow analysis ... 36
2.1. 0th order data-flow reaching ... 36
2.2. 0th order data-flow reaching rules (1) ... 36
2.3. 0th data-flow reaching rules (2) ... 36
2.4. Definition of 0th order data-flow reaching ... 36
2.5. Applications of 0th order data-flow reaching ... 37
2.6. Definition of 0th order compact forward data-flow relation ... 37
2.7. Definition of 0th order compact backward data-flow relation ... 37
3. 0th order DFG and reaching example ... 37
3.1. Example module ... 37
3.2. DFG for `dataflow` module ... 38
3.3. Applying data-flow reaching rules (1) ... 38
3.4. Applying data-flow reaching rules (2) ... 38
3.5. Applying data-flow reaching rules (3) ... 38
3.6. Applying data-flow reaching rules (4) ... 38
3.7. Applying data-flow reaching rules (5) ... 39
3.8. Applying data-flow reaching rules (6) ... 39
3.9. Applying data-flow reaching rules (7) ... 39
3.10. Applying data-flow reaching rules (8) ... 39
3.11. Applying data-flow reaching rules (9) ... 39
8. Lecture 8 ... 40

Created by XMLmind XSL-FO Converter.

1. Control-Flow ... 40
1.1. Control-Flow Analysis ... 40
1.2. Control-Flow Graph in general ... 40
1.3. Control-Flow Graph for Erlang ... 40
1.4. Notations in rules ... 40
1.5. Control-flow rule: Unary operator ... 40
1.6. Unary operator – Example ... 41
1.7. Control-flow rule: Left associative operator ... 41
1.8. Left associative operator – Example ... 41
1.9. Control-flow rule: Right associative operator ... 41
1.10. Control-flow rule: Comparison operator ... 41
1.11. Control-flow rule: Andalso operator ... 41
1.12. Andalso operator – Example ... 42
1.13. Control-flow rule: Orelse operator ... 42
1.14. Control-flow rule: Send operator ... 42
1.15. Control-flow rule: Parenthesis ... 42
1.16. Control-flow rule: Tuple expression ... 42
1.17. Control-flow rule: List expression ... 43
1.18. Control-flow rule: List comprehension (1) ... 43
1.19. Control-flow rule: List comprehension (2) ... 43
1.20. Control-flow rule: List comprehension (3) ... 43
1.21. List comprehension – Example ... 44
1.22. Control-flow rule: Function application ... 44
1.23. Function application – Example ... 44
1.24. Function definition ... 45
1.25. Control-flow rule: Function definition ... 45
1.26. Function – Example ... 46
1.27. Case expression ... 46
1.28. Control-flow rule: Case expression ... 46
1.29. Receive expression ... 47
1.30. Control-flow rule: Receive expression ... 47
9. Lecture 9 ... 49
1. Dominators and Postdominators ... 49
1.1. Dominators ... 49
1.2. Immediate Dominance ... 49
1.3. Postdominators ... 49
1.4. Postdominator calculating algorithm ... 49
1.5. Postdominator calculating algorithm (cont.) ... 49
1.6. Calculating Immediate Postdominator ... 50
1.7. Example: CFG ... 50
1.8. Example: CFG ... 52
1.9. Example: Postdominators ... 54
1.10. Example: Postdominators ... 56
1.11. Example: Postdominators ... 58
1.12. Example: Postdominators ... 60
1.13. Example: Postdominators ... 62
1.14. Example: Postdominators ... 64
1.15. Example: Postdominators ... 66
1.16. Example: Postdominators ... 68
1.17. Example: Postdominators ... 70
1.18. Example: Postdominators ... 72
1.19. Example: Postdominators ... 74
1.20. Example: Postdominators ... 76
1.21. Example: Postdominators ... 78
1.22. Example: Postdominators ... 80
1.23. Example: Postdominators ... 82
1.24. Example: Postdominators ... 84
1.25. Example: Postdominators ... 86
1.26. Example: Postdominators ... 88
1.27. Example: Postdominators ... 90

Created by XMLmind XSL-FO Converter.

1.28. Example: Remarks on Postdominator Calculation ... 92
1.29. Example: Immediate Postdominators ... 92
1.30. Example: Immediate Postdominators ... 92
1.31. Example: Immediate Postdominators ... 92
1.32. Example: Immediate Postdominators ... 93
1.33. Example: Immediate Postdominators ... 93
1.34. Example: Immediate Postdominators ... 94
1.35. Example: Immediate Postdominators ... 94
1.36. Example: Immediate Postdominators ... 95
1.37. Example: Immediate Postdominators ... 96
1.38. Example: Immediate Postdominators ... 96
1.39. Example: Immediate Postdominators ... 97
1.40. Example: Immediate Postdominators ... 97
1.41. Example: Immediate Postdominators ... 98
1.42. Example: Postdominator Tree ... 98
10. Lecture 10 ... 100
1. Control-dependence ... 100
1.1. Control-Dependence Analysis ... 100
1.2. Control-Dependence Analysis ... 100
1.3. Example (1) ... 100
1.4. Example (2) ... 102
1.5. Example (3) ... 104
1.6. Example (4) ... 106
1.7. Control Dependence Calculating Algorithm ... 107
1.8. Extending the Control Dependence Calculating Algorithm (1) ... 107
1.9. Example (CFG of Factorial Function) ... 107
1.10. Example (Simple CDG) ... 110
1.11. Example (Composed CDG) ... 110
2. Dependence Graph ... 110
2.1. Extending the Control Dependence Graph ... 110
2.2. Data Dependence ... 111
2.3. Further Dependencies ... 111
11. Lecture 11 ... 112
1. First order data-flow ... 112
1.1. Extended example module ... 112
1.2. 0th order DFG for the extended `dataflow` module ... 112
1.3. Problems with the 0th order data-flow analysis ... 112
1.4. 1st order data-flow analysis ... 113
1.5. Extending the data-flow rules ... 113
1.6. 1th order DFG for the extended `dataflow` module ... 113
1.7. Formal rule for the function call ... 113
1.8. Deriving from the 0th order data-flow rules(1) ... 114
1.9. Deriving from the 0th order data-flow rules(2) ... 114
1.10. Deriving from the 0th order data-flow rules(3) ... 114
1.11. Notations for `Definition 4` ... 115
1.12. Definition 4 (1) ... 115
1.13. Definition 4 (2) ... 115
2. Higher order data-flow analysis ... 116
2.1. N^{th} Order Analysis ... 116
2.2. Why generalisation is required? ... 116
2.3. Why generalisation is required? ... 116
12. Lecture 12 ... 117
1. Concurrent data-flow ... 117
1.1. Message passing ... 117
1.2. Processes and Message Passing ... 117
1.3. Concurrent data-flow analysis ... 117
1.4. Detecting Spawned Processes ... 117
1.5. Example ... 118

Created by XMLmind XSL-FO Converter.

1.6. Process analysis 118
1.7. Function Analysis 118
1.8. Example(1) 118
1.9. Example(2) 119
1.10. Detecting registered processes 119
1.11. Calculating values for a *register* call 119
1.12. Calculating possible functions 119
1.13. Modified example(1) 120
1.14. Modified example(2) 120
1.15. Heuristics 120
1.16. Heuristics based on the partial knowledge (1) 120
1.17. Heuristics based on the partial knowledge (2) 121
1.18. Example (1) 121
1.19. Example (2) 121
1.20. Possible message recipient at sender side (1) 121
1.21. Possible message recipient at sender side (2) 122
1.22. Analysis of receivers 122
1.23. Concurrent data-flow rule 122
1.24. Connection between send and receive sides 123
1.25. Extending the previous example (1) 123
1.26. Extending the previous example (2) 123
1.27. Refining the analysis 124
1.27.1. Improving the 1st Order Data-Flow Analysis 124
1.28. Refining the 1st Order Data-Flow Analysis 124
1.29. Example (1) 124
1.30. Example (2) 124
13. Lecture 13 126
1. Considered language elements 126
1.1. Examined Language Constructs 126
1.2. ETS tables 126
2. Communication Model 126
2.1. Representation 126
2.2. Non trivial steps... 126
2.3. The Magic Behind the Steps 127
3. Motivating Example 127
4. Algorithm 128
4.1. Process Identification 128
4.2. Process Nodes 129
4.3. Process Nodes 130
4.4. Process Nodes 132
4.5. Process Communication 132
4.6. Process Communication 134
4.7. Process Communication 135
4.8. Hidden Communication 136
4.9. Hidden Process Nodes 137
4.10. Hidden Communication 140
4.11. Process Relations 142
5. Algorithm Description 142
5.1. Identifying Processes 142
5.2. Identifying Processes 142
5.3. Creating Process Nodes 142
5.4. Calculating Communication Edges 143
5.5. Calculating Hidden Dependencies 143
5.6. Calculating Hidden Dependencies 143
5.7. Calculating write edges 144
5.8. Calculating write edges 144
5.9. Calculating read edges 144
14. Lecture 14 145
1. Static Analysis Tools for Erlang 145
1.1. Erlang Tools 145

Created by XMLmind XSL-FO Converter.

- 2. RefactorErl ... 145
  - 2.1. Source code analysis and transformation ... 145
  - 2.2. Source code analysis and transformation ... 145
  - 2.3. Demo ... 146
  - 2.4. Demo ... 146
- 3. Wrangler ... 147
  - 3.1. Wrangler as a Refactoring Tool ... 147
  - 3.2. Demo ... 147
- 4. Dialyzer & Typer & Tidier ... 148
  - 4.1. DIscrepancy AnalYZer for ERlang programs ... 148
  - 4.2. The Tidier Refactoring Tool ... 148
  - 4.3. Demo ... 149
- 5. Other Tools ... 149
  - 5.1. Outside the Erlang World ... 149
  - 5.2. Outside the Erlang World ... 149
  - 5.3. Why is it important? ... 150

15. Practice 1 ... 151
- 1. Introduction ... 151
  - 1.1. Functional Programming ... 151
  - 1.2. Properties ... 151
- 2. Erlang ... 151
  - 2.1. Erlang – Properties ... 151
  - 2.2. When To Use Erlang? ... 152
  - 2.3. Erlang shell ... 152
  - 2.4. Useful Shell Commands ... 152
- 3. Language constructs ... 152
  - 3.1. Terms ... 152
  - 3.2. Comparison Of Types ... 153
  - 3.3. Arithmetic, Bit and Logical operators ... 153
  - 3.4. Variables and Pattern Matching ... 153
  - 3.5. Modules ... 153
  - 3.6. Attributes ... 154
  - 3.7. Functions – ModName:FunName/Arity ... 154
  - 3.8. Built In Functions (BIF) ... 154
  - 3.9. Lists ... 154
  - 3.10. Conditional Evaluation – case, if ... 154
  - 3.11. Guard Expressions ... 155
  - 3.12. Fun Expressions ... 155
  - 3.13. Dynamic constructs ... 155
  - 3.14. Trapping Run-time Errors ... 155
  - 3.15. Records ... 156
  - 3.16. Macros ... 156

16. Practice 2 ... 157
- 1. Concurrent Erlang ... 157
  - 1.1. Processes ... 157
  - 1.2. Example Ping-Pong Server ... 157
  - 1.3. Concurrent language elements ... 157
  - 1.4. Process links and error handling ... 157
  - 1.5. Registering processes ... 158
  - 1.6. Erlang Term Storage – ETS(1) ... 158
  - 1.7. Erlang Term Storage – ETS(2) ... 158
- 2. Distributed Erlang ... 158
  - 2.1. Distributed Erlang Nodes ... 158
- 3. Advanced topics ... 159
  - 3.1. Ports and Port Drivers ... 159
  - 3.2. Connection with other languages ... 159
  - 3.3. Nice features ... 159

17. Practice 3 ... 161
- 1. RefactorErl ... 161
  - 1.1. History ... 161

Created by XMLmind XSL-FO Converter.

1.2. Motivation ... 161
1.3. Our solution ... 162
1.4. Requirements ... 162
1.5. Design goals ... 162
1.6. Emerged research topic ... 163
2. Architecture ... 163
2.1. Three-layered graph model ... 163
2.2. Example graph for `add/2` ... 164
2.3. Graph storage ... 164
2.4. Other details ... 165
3. Features ... 165
3.1. Features ... 165
3.2. User Interfaces ... 165
3.3. Web UI ... 166
3.4. Where to find us? ... 166
4. Install & Configure ... 166
4.1. Installation and configuration ... 166
4.2. Installation and configuration ... 166
4.3. Starting the tool ... 167
4.4. Start Options ... 167
4.5. Start Options ... 167
4.6. Start Options ... 167
18. Practice 4 ... 169
1. Analysing Erlang Modules ... 169
1.1. Building the Database ... 169
2. The Semantic Program Graph ... 169
2.1. Three-layered graph model ... 169
2.2. Lexical Schema ... 169
2.3. Syntactic Schema ... 170
2.4. Syntactic Schema ... 170
2.5. Semantic Schema ... 170
2.6. Semantic Schema ... 171
2.7. Semantic Schema ... 171
2.8. Simple Erlang File ... 171
2.9. Example graph for `add/2` ... 171
2.10. Reproduce the Original Source File ... 172
3. Graph Traversal ... 172
3.1. Path expressions ... 172
3.2. Path expression example ... 173
3.3. Exercises ... 173
3.4. Exercises ... 173
3.5. Exercises ... 174
3.6. Query Library ... 174
3.7. Query example ... 174
3.8. Exercises ... 174
3.9. Exercises ... 175
3.10. Exercises ... 175
3.11. Exercises ... 175
19. Practice 5 ... 177
1. Language Definition ... 177
1.1. Semantic query language ... 177
1.2. Syntax of the queries ... 177
1.3. Syntax of the queries ... 177
2. Language Elements ... 178
2.1. Entities ... 178
2.2. Initial Selectors ... 178
2.3. File Selectors ... 179
2.4. File Properties ... 179
2.5. Function Selectors ... 179
2.6. Function Properties ... 180

Created by XMLmind XSL-FO Converter.

2.7. Function Clause Selectors ... 180
2.8. Function Clause Properties ... 181
2.9. Expression Selectors ... 181
2.10. Expression Properties ... 182
2.11. Variable Selectors ... 182
2.12. Variable Properties ... 182
2.13. Record Selectors ... 182
2.14. Record Properties ... 183
2.15. Record Field Selectors ... 183
2.16. Record Field Properties ... 183
2.17. Macro Selectors ... 183
2.18. Macro Properties ... 183
2.19. Statistics ... 184
3. Usage ... 184
3.1. Semantic query examples ... 184
3.2. Semantic query examples ... 184
3.3. Exercises ... 185
20. Practice 6 ... 186
1. Reminder ... 186
1.1. Syntax of the Semantic Queries ... 186
1.2. Semantic Queries ... 186
1.3. Semantic Query Examples ... 186
2. Use Cases ... 186
2.1. Finding Functions and References ... 186
2.2. Finding Functions and References ... 187
2.3. Finding Records, Record Fields and References ... 187
2.4. Atom references ... 187
2.5. String References ... 188
2.6. An Advanced Query for Records ... 188
2.7. Detecting the Possible Values of a Variable ... 188
2.8. Detecting Dynamic Function Calls ... 188
2.9. Defining "Dynamic” Function References ... 189
2.10. Defining "Dynamic” Function References ... 189
2.11. Finding function calls ... 189
2.12. Finding function calls ... 189
2.13. Calculating Macro Values ... 190
21. Practice 7 ... 191
1. Structural Complexity Metrics ... 191
1.1. Complexity metrics ... 191
1.2. Metrics In RefactorErl ... 191
1.3. Metric Query Language Examples ... 191
1.4. Metric Query Language Examples ... 191
2. Metrics as Semantic Query Properties ... 191
2.1. Software Metrics ... 191
2.2. File Metrics ... 192
2.3. Function Metrics ... 193
2.4. Function Clause Metrics ... 193
3. Checking Coding Conventions ... 194
3.1. Coding Convention Rules ... 194
4. Metric Mode ... 195
4.1. Metric Mode ... 195
5. Exercises ... 195
5.1. Build Queries! ... 195
22. Practice 8 ... 196
1. Dependency Analysis ... 196
1.1. Dependency analysis ... 196
1.2. Types of dependency analysis ... 196
1.3. Module dependencies ... 196
1.4. Function dependencies ... 196
1.5. “Function-block” dependencies ... 196

Created by XMLmind XSL-FO Converter.

2. Usage ... 196
2.1. Function/module dependency analysis in ri ... 196
2.2. Parameters ... 196
2.3. Parameters ... 197
2.4. Smart graph ... 197
2.5. Examples ... 198
2.6. Function-block analysis in ri ... 198
2.7. Function-block analysis in ri ... 198
2.8. Examples ... 199
2.9. Function-block analysis in ri ... 199
2.10. Function-block analysis in ri ... 199
2.11. Examples ... 200
2.12. Examples ... 200
2.13. Checking Layers in ri ... 200
2.14. Checking Layers in ri ... 200
2.15. Examples ... 200
2.16. Exercise ... 201
23. Practice 9 ... 202
1. Clustering ... 202
1.1. Motivation ... 202
1.2. Clustering ... 202
1.3. Types of clustering in RefactorErl ... 202
2. Usage in RefactorErl ... 202
2.1. Parameters for clustering ... 202
2.2. Output formats ... 203
2.3. Parameters for agglomerative clustering ... 203
2.4. Parameters for agglomerative clustering ... 203
2.5. Parameters for agglomerative clustering ... 203
2.6. Parameters for genetic clustering ... 203
2.7. Parameters for genetic clustering ... 204
2.8. Parameters for decomposition ... 204
2.9. Running the clustering on Mnesia! ... 204
2.10. Running the clustering on Mnesia! ... 204
2.11. Running the clustering on Mnesia! ... 205
2.12. Running the clustering on Mnesia! ... 205
2.13. Exercise ... 205
24. Practice 10 ... 206
1. Refactoring ... 206
1.1. Refactoring with RefactorErl ... 206
1.2. Refactoring steps ... 206
1.3. Refactoring steps ... 206
2. Rename Refactorings ... 207
2.1. Rename Variable ... 207
2.2. Rename X to Y ... 207
2.3. Rename Function ... 208
2.4. Rename Function ... 208
2.5. Rename doit to send_start ... 208
2.6. Rename Record ... 209
2.7. Renaming record "person" to "member" ... 209
2.8. ... 209
2.9. Renaming field name to id ... 210
2.10. ... 210
2.11. Renaming LessEq to Leq ... 211
2.12. ... 211
2.13. Renaming header file header1.hrl to newname ... 211
2.14. Rename module ... 212
2.15. Renaming module mod1 to newmod ... 212
3. Function Interface ... 213
3.1. Introduce Function Parameter/Generalize Function ... 213
3.2. Introduce Function Parameter/Generalize Function ... 213

Created by XMLmind XSL-FO Converter.

3.3. Introduce a new parameter to function double/1 214
3.4. Reorder function parameters 214
3.5. Reorder function parameters 214
3.6. Reordering parameters 215
3.7. Introduce tuple / Tuple function parameters 215
3.8. Introduce tuple / Tuple function parameters 215
3.9. Create a tuple from the arguments of step/2 216
3.10. Introduce Import List Element 216
3.11. Introduce Import List Element 216
3.12. Introducing an import list for lists:sort/1 217
4. Move Definitions 217
4.1. Move macro 217
4.2. Moving macro Person the header.hrl 217
4.3. Move record 218
4.4. Moving record msg to message.hrl 218
4.5. Move function 219
4.6. Moving pzip/1 to xlists.erl 219
5. Data Structure Related Refactorings 220
5.1. Introduce record 220
5.2. Introducing the record cart 220
5.3. 221
5.4. Upgrading regexp:match/2 221
6. Expression Structure 221
6.1. Eliminate Variable 221
6.2. Eliminate Variable 222
6.3. Eliminating the variable Y 222
6.4. Introduce variable/Merge subexpression duplicates 222
6.5. Introducing variable V 223
6.6. Inline Function 223
6.7. Inlining sort/1 223
6.8. Introduce function/Extract function 224
6.9. Introducing two_sol/3 224
6.10. Inline macro 225
6.11. Inlining ?Add(A,A) 225
6.12. Eliminate fun expression/Expand fun expression 226
6.13. Eliminating implicit reference to far:away/2 226
6.14. Introduce/eliminate list comprehensions 226
6.15. Transforming to lists:filter/2 227
6.16. Exercise 227
25. Practice 11 228
1. Refactoring with RefactorErl 228
1.1. Refactoring workflow 228
1.2. Transformation 228
1.3. Implementation 228
1.4. prepare/1 229
1.5. prepare/1 229
1.6. Transformations in general 229
1.7. Restrictions 229
1.8. Error Messages 229
2. Case Study: Rename Variable 230
2.1. Rename Variable 230
2.2. Rename X to Y 230
2.3. reftr_rename_var.erl 230
2.4. Querying the Arguments 230
2.5. Querying information to check the side conditions 231
2.6. Asking the new variable name 231
2.7. Performing the transformation 231
2.8. Performing the transformation 231
2.9. Side condition checking with interaction 232
2.10. Exercise 232

Created by XMLmind XSL-FO Converter.

26. Practice 12 233
1. Duplicated Code Detection 233
1.1. Code Duplicates 233
1.2. Duplicate Code Detectors 233
1.3. Clone IdentifiErl 233
1.4. Clone IdentifiErl 233
1.5. Clone IdentifiErl 234
1.6. Search Duplicates 234
1.7. Search Duplicates 235
1.8. Search Duplicates 235
1.9. Search Duplicates 235
1.10. Search Duplicates 236
1.11. Search Duplicates 236
1.12. Search Duplicates 236
2. Eliminating Code Clones with Refactorings 237
2.1. Using Refactorings 237
2.2. Eliminating clones 237
2.3. Generalize over the function call 237
2.4. Introducing the general check function 238
2.5. Change the call in check_2 238
2.6. Done! 238
2.7. Eliminate this clone! 239
2.8. Exercise 239
27. Practice 13 240
1. Data-flow Analysis Introduction 240
1.1. Data-flow 240
1.2. Reaching definition analysis 240
1.3. Kinds of Data-Flow edges 240
1.4. Data-Flow reaching 240
1.5. DFG in RefactorErl 241
1.6. Example Graph 241
1.7. Example Graph 241
1.8. Exercise 241
2. Reaching in RefactorErl 241
2.1. Semantic Queries 241
2.2. Reaching in refanal_dataflow.erl 242
2.3. Reaching in refanal_dataflow.erl 242
2.4. Exercise 242
28. Practice 14 243
1. Building the DB 243
1.1. Running example 243
1.2. Building the database 243
1.3. SPG of factorial 243
2. Control-Flow Graph 243
2.1. Calculating the Control Flow Graph 243
2.2. Calculating the Control Flow Graph – Managing the Server 244
2.3. Calculating the Control Flow Graph – Asynchronous communication 244
2.4. Calculating the Control Flow Graph – Synchronous communication 244
2.5. Example CFG 245
2.6. Exercise 247
3. Postdominator Tree 247
3.1. Calculating the Postdominator Tree 247
3.2. Example PDT 247
3.3. Exercise 247
4. Dependence Graph 248
4.1. Calculating the Control Dependence Graph 248
4.2. Calculating the Control Dependence Graph – Managing the Server 248
4.3. Calculating the Control Dependence Graph – Asynchronous communication 248
4.4. Calculating the Control Dependence Graph – Synchronous communication 248
4.5. Example CDG 249

Created by XMLmind XSL-FO Converter.

4.6. Example CCDG ... 249
4.7. Exercise ... 250
4.8. Calculating the Dependence Graph ... 250
4.9. Example DG ... 250
5. Exercises ... 251
5.1. Exercises ... 251

Created by XMLmind XSL-FO Converter.

Chapter 1. Lecture 1

1. Syllabus

1.1. Syllabus

- The functional programming language, Erlang
- The syntax of Erlang programs
- Symbol table
- Abstract Syntax Tree
- Program Graph
- Code generation
- Control-flow analysis, Control-Flow Graph
- Data-flow analysis, Data-Flow Graph
- Dependency analysis, Dependency Graph
- Program code and software model re-engineering

2. Static Analysis

2.1. Static Analysis

- Analysis of computer software that is performed without actually executing the programs built from that software
- Usage: vary from finding possible coding errors, checking coding conventions, visualisation of models, to formal methods that mathematically prove properties about a given program
- Intermediate source code representation is required
- Different levels of abstraction

2.2. Reverse Engineering of Software

- Special static analysis
- "Reverse engineering is the process of analyzing a subject system to create representations of the system at a higher level of abstraction." (Chikofsky, E.J.; J.H. Cross)
- "Going backwards through the development cycle" (Warden, R)

3. Introduction to Erlang

3.1. Functional Programming

- The topmost level is a set of modules
- The module is a set of declaration (type, class, function)
- Initial statement

Created by XMLmind XSL-FO Converter.

- Evaluation
- Based on mathematical model (Lambda Calculus)
- Turing complete

3.2. Properties

- Referential transparency
- (Static typing)
- Higher-order functions
- (Currying)
- Recursion
- Strict(/lazy) evaluation
- List comprehensions
- Pattern matching
- ("Offset rule")
- IO model

3.3. History

- 1982 - 1986 – Experiments with different programming languages
- 1987 – First experiments with Erlang
- 1988 - 1990 – Experiences with Erlang in telecom world
- 1993 – Distributed programming / First Erlang book (The BOOK)
- 1996 – OTP R1
- 1998 – Released as Open Source
- 2005 – R11 multicore

3.4. Erlang – Properties

- Declarative – Functional programming language, high level of abstraction
- Dynamically typed
- Concurrency – explicit concurrency, LWP
- Soft real-time characteristics
- Robustness – supervison trees
- Distribution – transparent, explicit, network
- Openness, external interfaces – "ports"
- Portability – Unix, Win., ... , heterogeneous network
- SMP Support – multicore

Created by XMLmind XSL-FO Converter.

- “Hot code loading”

3.5. Erlang – Ericsson Language

- Erlang, Agner Krarup (1878-1929)
- Danish mathematician
- Erlang formula
- erlang – unit of load on telephone circuits

3.6. When To Use Erlang?

- Complex, continuously operating, scalable, maintainable, distributed
- Rapid and efficient development
- Fault-tolerant (software, hardware) systems
- Hot-code loading

3.7. Who Uses Erlang?

- Ericsson – telecommunication (AXD301 ATM switch), simulation, testing, 3G, GPRS
- Amazon – Simple DB (DBMS)
- Yahoo – Online bookmarks service
- Facebook – chat server
- T-Mobile – SMS gateway
- Motorola – call processing
- MochiWeb – http server
- CouchDb – document database server (multicore, multiserver clusters)

Created by XMLmind XSL-FO Converter.

- YAWS – Yet Another Web Server
- Wings3D – 3D modeling
- and many other...

Created by XMLmind XSL-FO Converter.

Chapter 2. Lecture 2

1. The Syntax of Erlang programs

1.1. Language elements

- M – module
- F – function definition
  - G – guard
  - P – pattern
  - E – expression
- R – record definition
- MC – macro definition
- AT – attributes

1.2. Language elements – Examples

- M – `-module(mymod).`
- F – `f()-> ok.`
  - G – `N>0`
  - P – `[Head|Tail]`
  - E – `X + f(X,Y)`
- R – `-record(myrec, {myfield1, myfield2}).`
- MC – `-define(mymac, 42).`
- AT – `-myattr('myname: X. Y.').`

1.3. Constants and Variables

M ::=

V ::= variables (including the underscore pattern (_))

A ::= atoms

I ::= integers

K ::= A | I | other constants (e.g. string, float, char)

1.4. Constants and Variables – Examples

M:

V : VarName, _Varname, _, VARName01, etc

A : atom1, aTom1, 'atom again & again', etc

I : 1, 2, 3, -1, etc

K : $A \mid I \mid$ "Constant string", 0.1, $K, etc

1.5. Functions

M ::=

F ::= A (P,...,P) when G -> E,...,E ;

$\vdots$

A (P,...,P) when G -> E,...,E .

1.6. Functions – Example

```
factorial(0) ->
    1;
factorial(N) when N > 1 ->
    N*factorial(N-1).
```

1.7. Patterns

M ::=

P ::= K

$\mid V$

$\mid$ { P,...,P }

$\mid$ [P,...,P | P]

$\mid$ # A { $A = P$,...,$A = P$ }

$\mid P = P$

$\mid << BP,\ldots,BP >>$

BP ::= $P : P \mid P/P \mid P : P/P$

1.8. Patterns – Example

M ::=

P : constants, 1, 0.1, $K, "Constants"

VarName, _,

{ VarName, atom, int, 1}, {},

[Hed | Tail], [], [1,2 | VarTail],

#recname{field1=Var, field2=2},

Created by XMLmind XSL-FO Converter.

List = [Head | Tail],

«Var1, Var2», «Var1:4/binary, Var2», etc

1.9. Expressions 1.

M ::=

E ::= $K \mid V$

$\mid \{ E, \ldots, E \}$

$\mid E_{List}$

$\mid E \; op \; E$

$\mid op \; E$

$\mid P = E$

$\mid E (E, \ldots, E) \mid E : E (E, \ldots, E)$

$\mid E_b$

M ::=

E_{List} ::=

$[E, \ldots, E \mid E]$

$\mid [E \,||\, P \text{<-} E, \ldots, P \text{<-} E, E, \ldots, E]$

1.10. Expressions – List

```
  an_atom
Variable
{tuple1, tuple2}
Var = 1
A + B
not 2
A andalso B
{Var1, Var2} = {2,3}
mymod:myfun(par1, Par2)
[Elem1, Elem2, Elem3 | Tail]
[1,2,3 | [1,2,3]]
[X*X || X <- List, X > N]
```

1.11. Expressions 2.

M ::=

E_b ::=

case E of

P when G -> $E, \ldots, E$;

Created by XMLmind XSL-FO Converter.

$\vdots$

P when G -> $E, \ldots, E$

end

$|$ if

G -> $E, \ldots, E$;

$\ldots$

G -> $E, \ldots, E$

end

$| E_r$

1.12. Expressions – Branching

```
    case f(X) of
      [H|T] -> list;
      []    -> nil
  end

  if A > B   -> A;
     A < B   -> B;
     A  == B -> A
  end
```

1.13. Expressions 3.

M ::=

E_r ::=

receive

P when G -> $E, \ldots, E$;

$\vdots$

P when G -> $E, \ldots, E$

after

E -> $E, \ldots, E$

end

$|$ begin

$E, \ldots, E$

end

$| E_t$

1.14. Expressions – Branching

Created by XMLmind XSL-FO Converter.

```
  receive
    [H|T] -> list;
    []    -> nil
after
    10    -> timeout
end

begin
    Y = f(X),
    Y + X
end
```

1.15. Expressions 4.

M ::=

E_t ::=

try $E, \ldots, E$ of

P when G -> $E, \ldots, E$;

$\ldots$

P when G -> $E, \ldots, E$

catch

$P : P$ when G -> $E, \ldots, E$;

$\ldots$

$P : P$ when G -> $E, \ldots, E$

after

$E, \ldots, E$

end

$|$ catch E

$|$ E_f

1.16. Expressions – Branching

```
  try
    Y = f(X),
    g(Y, X)
of
    [H|T] -> list;
    []    -> nil
catch
    error:Reason -> Reason;
    _:_          -> nok
after
    do_sth()
end
```


Created by XMLmind XSL-FO Converter.

```
catch bad_fun(X)
```

1.17. Expressions 5.

M ::=

E_f ::=

fun

($P,\ldots,P$) when G -> $E,\ldots,E$;

$\ldots$

($P,\ldots,P$) when G -> $E,\ldots,E$

end

fun $A:A/I$

| E_{rec}

1.18. Expressions – Funexpressions

```
  fun
    (A, [H|T]) -> A + 1;
    (A, [])    -> A
end

fun mymod:myfun/2
```

1.19. Expressions 6.

M ::=

E_{rec} ::=

A { $A = E$,..., $A = E$ }

| E # A { $A = E$,..., $A = E$ }

| # A . A

| E # A . A

| E_{binary}

1.20. Expressions – Records

```
  Var = #person{firstname = "Melinda",
                lastname = "Toth"}
Var#person{firstname = "Melindaaaa"}
Var#person.firstname
#person.firstname
```


Created by XMLmind XSL-FO Converter.

1.21. Expressions 7.

M ::=

E_{binary} ::=

$<< E, \ldots, E >>$

1.22. Expressions – Binary

```
  <<A, B, C>>
<<1,2,3>>
<<1:4,2:6,3:4>>
<<A/binary, B:4/unsigned-integer>>
```

1.23. Guards

- Expressions
- Restricted to side effect free operations
- No user defined function calls

Created by XMLmind XSL-FO Converter.

Chapter 3. Lecture 3

1. Abstract syntax tree

1.1. Abstract syntax tree (AST)

- Output of the syntax analyser
- Representation of the syntactic structure
- Nodes represent constructs in the source
- Lack of some information according to the real syntax (e.g. parentheses, semicolons, whitespaces, etc.)

1.2. Abstract syntax tree (AST)

- Used by compilers (optimisation, code generation)
- Input of the semantic analysis
- Annotated abstract syntax tree (line information, additional semantic information)

1.3. About RefactorErl

- RefactorErl is a static source code analyser tool
- Representing the source code as a Semantic Program Graph (AST + Semantic information)
- Lexer + Parser + Preprocessor + Semantic analysis

1.4. AST in the RefactorErl

- Description of the syntax of *Erlang* – refcore_erlang.syntax
- The lexer is generated from this syntax description using the *leex*
- The parser is generated from this syntax description using the *yecc*

1.5. Parser in RefactorErl

- Layout preserving
- Macro syntax support
- Source can be restored from AST

1.6. Rule syntax

```
  Ruleset  ::= Name '->' Rule { '|' Rule}
Rule     ::= Name | Data '(' Children ')
Data     ::= '#' Class '{' Attrib { ',' Attrib } '}'
Attrib   ::= atom '=' Value | atom '<-' Token
Children ::= Child { Child }
Child    ::= Token | Link '->' Name | '{' Children '}' | '[' Children ']'
Name     ::= variable
Token    ::= atom
Link     ::= atom
Value    ::= atom | integer | string
```


Created by XMLmind XSL-FO Converter.

1.7. Module attribute

```
FModule ->
#form{type=module, paren=default,
      tag<-'atom'}
  ('-' 'module' '(' 'atom' ')' 'stop')
| #form{type=module, paren=no,
        tag<-'atom'}
  ('-' 'module'     'atom'     'stop')
```

1.8. Module attribute – Example

```
FModule ->
#form{type=module, paren=default,
      tag<-'atom'}
  ('-' 'module' '(' 'atom' ')' 'stop')

    -module(mymod).
```

1.9. Export attribute

```
FExport ->
#form{type=export, paren=default}
  ('-' 'export' '('
    eattr->EAFunList ')' 'stop')

| #form{type=export, paren=no}
    ('-' 'export'
      eattr->EAFunList     'stop')
```

1.10. Export attribute – Example

```
FExport ->
#form{type=export, paren=default}
  ('-' 'export' '('
    eattr->EAFunList ')' 'stop')

    -export([f:1, g/2]).
```

1.11. Import attribute

```
FImport ->
#form{type=import, paren=default}
  ('-' 'import' '('
    eattr->EAtom ','
      eattr->EAFunList ')' 'stop')
| #form{type=import, paren=no}
    ('-' 'import'
      eattr->EAtom ','
        eattr->EAFunList     'stop')
```

1.12. Record definition

```
FRecord ->
```


Created by XMLmind XSL-FO Converter.

```
#form{type=record, paren=default,
      tag<-'atom'}
   ('-' 'record' '(' 'atom' ','
     '{' [tattr->TFldSpec {','
          tattr->TFldSpec}] '}' ')' 'stop')

| #form{type=record, paren=no,
        tag<-'atom'}
   ('-' 'record' 'atom' ','
     '{' [tattr->TFldSpec {','
          tattr->TFldSpec}] '}' 'stop')
```

1.13. Record definition – Example

```
FRecord ->
#form{type=record, paren=default,
      tag<-'atom'}
   ('-' 'record' '(' 'atom' ','
     '{' [tattr->TFldSpec {','
          tattr->TFldSpec}] '}' ')' 'stop')

    -record(myrec, {f1 = value, f2, f3})
```

1.14. Function

```
  FFunction ->
  #form{type=func}
     ( funcl->CFunction
       { ';' funcl->CFunction } 'stop' )

CFunction ->
  #clause{type=fundef}
     ( name->EAtom
       '(' [pattern->Expr {',' pattern->Expr}] ')'
         ['when' guard->Guards]
           '->' body->Expr {',' body->Expr})
```

1.15. Function Clause – Example

```
CFunction ->
#clause{type=fundef}
   ( name->EAtom
     '(' [pattern->Expr {',' pattern->Expr}] ')'
       ['when' guard->Guards]
         '->' body->Expr {',' body->Expr})
    f
     (  X, Y)
        when X > Y
         -> X + Y, ok
```

1.16. Case expression

```
ECase ->
#expr{type=case_expr}
   ('case' headcl->CExp 'of'
     exprcl->CPattern
       {';' exprcl->CPattern} 'end')
```

1.17. Case expression – Example

```
ECase ->
#expr{type=case_expr}
  ('case' headcl->CExp 'of'
    exprcl->CPattern
      {';' exprcl->CPattern} 'end')

    case something() of
       Pattern1 -> todo1();
       _        -> todo2()
    end
```

1.18. If expression

```
EIf ->
#expr{type=if_expr}
  ('if' exprcl->CGrd
     {';' exprcl->CGrd} 'end')
```

1.19. Receive expression

```
EReceive ->
#expr{type=receive_expr}
  ('receive' [exprcl->CPattern
                {';' exprcl->CPattern}]
             ['after' aftercl->CAfter]
   'end')
```

2. Symbol table

2.1. Symbol table

- Data structure (tree, hash table, lists, etc.)
  - identifiers from source
  - associated information (type, scope, address, etc.)
- Local symbol table (procedure, function)
- Global symbol table (module, entire program)

Created by XMLmind XSL-FO Converter.

Chapter 4. Lecture 4

1. Preprocessing

1.1. Preprocessor

- Resolves include file dependencies
- Substitutes macro applications
- Deals with conditional compilation
- Runs before building the AST building
- The semantic analysis evaluated on the preprocessed AST

2. Semantic Program Graph

2.1. Semantic graph model

Consists of three layers:

- *Lexical layer* – token list
- *Syntactic layer* – syntax tree
- *Semantic layer* – semantic layer, indirect relations between semantic and syntactic nodes

2.2. Semantic graph

- Abstract data type
- Representation of syntactic and semantic structure of the source code
- Based on the AST
- Query language for efficient information retrieval (path expressions)
- Nodes and links
- Special `root` node

2.3. Mathematical model

$$\mathcal{SG} = (N, A_N, A_V, A, T, E),$$

where

- N is the set of graph nodes, these will represent the nodes of the syntax tree and additional semantic nodes,
- A_N is a set of attribute names,
- A_V is a set of possible attribute values,
- $A : N \times A_N \to A_V$ is the node labeling partial function,
- T is a set of edge tags, and

Created by XMLmind XSL-FO Converter.

- $E : N \times T \times \mathbb{N}_0 \to N$ is a partial function that describes labeled, ordered edges between the nodes.

2.4. Graph Schema

$\mathcal{SCH} = (C, N_c, A_c, E_c)$,

where

- C is the set of permitted node class names.
- $N_c : N \to C$ is a total function that classifies nodes.
- $A_c \subseteq C \times A_n$ is a relation that contains the attributes that are used for a given class of nodes.
- $E_c : C \times T \to C$ is a partial function that describes valid edge tags between node classes.

2.5. Graph corresponds to the given Schema

A given graph $\mathcal{SG}$ is valid if the following applies with the schema $\mathcal{SCH}$:

- $\mathcal{D}_A = \{(n, a) \mid (N_c(n), a) \in A_c\}$ the attributes of a given node is the same defined for its class in the schema
- $\forall n, t, i : N_c(E(n, t, i)) = E_c(N_c(n), t)$ all links created are the ones permitted by the schema.

2.6. Graph traversal (`path expression`)

Definition for `path expressions`:

$$\begin{aligned} \mathcal{P} &= [PE_1, PE_2, \ldots, PE_k] \\ PE_i &= (t_i, d_i, f_i) \end{aligned}$$

where

- $t_i \in T$ is a link tag.
- $d_i \in \{F, B\}$ is a direction specifier (F stands for forward and B stands for backward).
- $f_i : \mathbb{N}_0 \times N \to \mathbb{L}$ is a filtering function

2.7. Graph traversal (`path expression` evaluation)

Evaluating `path expressions`:

$$\begin{aligned} \mathcal{E}(n, []) &= [n] \\ \mathcal{E}(n, [PE_1, PE_2, \ldots, PE_k]) &= \mathcal{E}(\mathcal{E}(n, PE_1), [PE_2, \ldots, PE_k]) \\ \mathcal{E}([n_1, \ldots, n_k], P) &= \mathcal{E}(n_1, P) + \ldots + \mathcal{E}(n_k, P) \\ \mathcal{E}(n, (t, F, f)) &= \{n' \mid n' = E(n, t, i) \wedge f(i, n') = true\} \\ \mathcal{E}(n, (t, B, f)) &= \{n' \mid E(n', t, i) = n \wedge f(0, n') = true\} \end{aligned}$$

2.8. Graph traversal (filtering the result)

Filtering the result within the traversal:

- TrueFilter: $true(i, n) = true$,

Created by XMLmind XSL-FO Converter.

- IndexFilter: $index(j)(i,n) = (i = j)$,
- RangeFilter: $index(b,e)(i,n) = (b \leq i \wedge i < e)$,
- IntersectionFilter: $intersect(n',t')(i,n) = (n \in \mathcal{E}(n',[(t',F,true)]))$, and
- AttributeFilter: $attribute(a,\odot,v)(i,n) = (A(n,a) \odot v)$, where $\odot$ can be $=$, $\neq$, $<$, $>$, $\leq$, and $\geq$.

2.9. Graph traversal (additional functions)

Additional functions:

- $last(n,t) = max\{i \mid (n,t,i) \in \mathcal{D}_E\}$
- and, or, and not logical operations for the $attribute$ function

2.10. Examples

```
  [file, form]
[{form, back}, {file, back}]
[file,
 {form, {index, '==', 4},
 funcl,
 {visib, back}]
```


Created by XMLmind XSL-FO Converter.

Chapter 5. Lecture 5

1. Introduction

1.1. Architecture of RefactorErl

- Parser adopted for static analysis purposes
- Semantic analyser modules
- Semantic graph model
- Generic graph model
- Storage model

2. Lexical layer

2.1. Lexical Schema

```
   -define(LEXICAL_SCHEMA,
         [{lex,     record_info(fields, lex),
                    [{mref, form},
                     {orig, lex}, {llex, lex}]},
          {file,    [{incl, file}]},
          {form,    [{iref, file}, {flex, lex},
                     {forig, form}, {fdep, form}]},
          {clause, [{clex, lex}]},
          {expr,    [{elex, lex}]},
          {typexp, [{tlex, lex}]}
         ]).

 -record(lex,       {type, data}).
 -record(token,     {type, text, prews="",
                     postws="", scalar, linecol}).
```

2.2. Lexical information

A *lexical* node created for each lexical element: `{'$gn', lex, int()}`

- The attributes for the node are: `type`, `tag`
- Linked to
  - the containing form (`flex`), clause (`clex`), expression (`elex`), type expression (`tlex`)
  - the original macro application (`orig, llex`)
- The lexical schema contains the preprocessor generated information: `iref, forig, fdep`

2.3. Token information

- The token stores the information about a lexical element
- The attributes of the tokens are: `type`, `text`, `prews`, `postws`, `scalar`, `linecol`

3. Syntactic layer

Created by XMLmind XSL-FO Converter.

3.1. Syntactic Schema

```
-define(SYNTAX_SCHEMA,
       [{root,   [],
                 [{file, file}]},
        {file,   record_info(fields, file),
                 [{form, form}]},
        {clause, record_info(fields, clause),
                 [{body, expr}, {guard, expr},
                  {name, expr},
                  {pattern, expr}, {tmout, expr}]},
        {expr,   record_info(fields, expr),
                 [{aftercl, clause}, {catchcl, clause},
                  {esub, expr},
                  {exprcl, clause}, {headcl, clause}]},
        {form,   record_info(fields, form),
                 [{eattr, expr}, {funcl, clause},
                  {tattr, typexp}]},
        {typexp, record_info(fields, typexp),
                 [{texpr, expr}, {tsub, typexp}]}]).
```

3.2. Syntactic Schema

```
 -record(file,     {type, path, eol, lastmod, hash}).
-record(form,     {type, tag, paren=default,
                   pp=none, hash, form_length,
                   start_scalar, start_line}).
-record(clause,   {type, var, pp=none}).
-record(expr,     {type, role, value, pp=none}).
-record(typexp,   {type, tag}).
```

3.3. File information

The *file* node represents the Erlang modules and headers: `{'$gn', file, int()}`

- The attributes for the node are: `type`, `path`, `eol`, `lastmod`, `hash`
- Linked to
  - the 'root' node (`file`)
  - the contained forms (`form`)

3.4. Form information

The *form* node represents the forms of the module: `{'$gn', form, int()}`

- The attributes for the node are: `type`, `tag`, `paren`, `pp`, `hash`, `form_length`, `start_scalar`, `start_line`
- Linked to its
  - attributes (`eattr, tattr`)
  - clauses (`funcl`)

3.5. Clause information

The *clause* node represent function and expression clauses: `{'$gn', clause, int()}`

- The attributes for the node are: `type`, `var`, `pp`

Created by XMLmind XSL-FO Converter.

- Linked to
  - the contained toplevel expressions (`body`), guards (`guard`) and patterns (`pattern`)
  - the name of the function (`name`)
  - the expression representing a timeout (`tmout`)

3.6. Expression information

The *expr* nodes represent the expressions: `{'$gn', expr, int()}`

- The attributes for the node are: `type`, `role`, `value`, `pp`
- Linked to
  - its clauses (`aftercl, catchcl, exprcl, headcl`)
  - its subexpressions (`esub`)

3.7. Type expression information

The *typexp* nodes represent the type information: `{'$gn', typexp, int()}`

- The attributes for the node are: `type`, `tag`
- Linked to
  - the referred expression (`texpr`)
  - the contained subtype information (`tsub`)

4. Semantic layer

4.1. Semantic Schema

```
 refanal_mod:schema/0
    [{module, record_info(fields, module),
              []},
     {root,   [{module, module}]},
     {file,   [{moddef, module}]},
     {clause, [{modctx, module}]}
    ]
refanal_fun:schema/0
    [func,    record_info(fields, func),
              [{funcall, func}, {dyncall, func},
               {ambcall, func}, {may_be, func}]},
     {form,   [{fundef, func}]},
     {clause, [{functx, clause}]},
     {expr,   [{modref, module}, {funeref, func}, {funlref, func},
               {dynfuneref, func}, {ambfuneref, func},
               {dynfunlref, func}, {ambfunlref, func},
               {localfundef, func}]},
     {module, [{func, func}, {funexp, func}, {funimp, func}]}
    ]
refanal_ets:schema/0
    [{ets_tab,record_info(fields, ets_tab),
              [{ets_ref, expr}, {ets_def, expr}]}]
```

4.2. Semantic Schema

Created by XMLmind XSL-FO Converter.

```
refanal_var:schema/0
    [{variable, record_info(fields, variable),
                [{varintro, expr}]},
     {clause,   [{scope, clause}, {visib, expr},
                 {vardef, variable}, {varvis, variable}]},
     {expr,     [{varref, variable}, {varbind, variable}]}
    ]
refanal_expr:schema/0
    [{expr, [{top, expr}, {clause, clause}]}]
refanal_rec:schema/0
    [{field,  record_info(fields, field),
              []},
     {typexp, [{fielddef, field}]},
     {expr,   [{fieldref, field}]},
     {record, record_info(fields, record),
              [{field, field}]},
     {file,   [{record, record}]},
     {form,   [{recdef, record}]},
     {expr,   [{recref, record}]}
    ].
```

4.3. Semantic Schema

```
 -record(module,    {name}).
-record(record,    {name}).
-record(field,     {name}).
-record(func,      {name              :: atom(),
                    arity             :: integer(),
                    dirty   = int     :: no | int | ext,
                    type    = regular :: regular | anonymous,
                    opaque  = false   :: false | module |
                                         name | arity}).
-record(variable,  {name}).
-record(env,       {name, value}).
-record(ets_tab,   {names}).
-record(pid,       {reg_name}).
```

4.4. Module information

A separate semantic node is added for every module: `{'$gn', module, int()}`

- The only attribute is the `name` of the module
- Linked to
  - the root node with `module` tag
  - the file node with `moddef` tag
  - every scope clause with `modctx` tag
  - every expression that explicitly refers to the module with `modref` tag

4.5. Function information

A separate semantic node is added for every function: `{'$gn', func, int()}`

- Attributes for the node are: `name`, `arity`, `dirty`, `type`, `opaque`
- Linked to

Created by XMLmind XSL-FO Converter.

- the defining module with `func` tag
- the function definition with `fundef` tag
- the module node with `funexp` tag, if the function is exported
- the module node with `funimp` tag, if the module imports the function
- cont...

4.6. Function information

- Attributes for the node are: `name`, `arity`, `dirty`, `type`, `opaque`
- Linked to
  - the defining module with `func` tag
  - the function definition with `fundef` tag
  - the module node with `funexp` tag, if the function is exported
  - the module node with `funimp` tag, if the module imports the function
  - every expression that explicitly refers to the function with `funlref` or `funeref` tag
  - every expression that dynamically or ambiguously refers to the function with `dynfunlref`, `dynfuneref`, `ambfunlref` or`ambfuneref` tag
  - every semantic function that refers to the function (`funcall, dyncall, ambcall, may_be`)
  - the clauses of the defined function (`functx`)

4.7. Variable information

A separate semantic node is added for every variable: `{'$gn', variable, int()}`

- The only attribute for the node is `name`.
- Linked to
  - the scope clause (function, list comprehension) with `variable` tag
  - every clause where variable is visible with `varvis` tag
  - every top-level expression that introduces the variable with `varintro` tag
  - every variable expression that bind a value to the variable with `varbind` tag
  - every variable expression that explicitly refers to the variable (reads) with `varref` tag

4.8. Context information

- *top* – expression to a top-level expression
- *scope* – containing clause
- *clause* – directly contained clauses
- *visib* – top-level expressions in clauses

4.9. Record information

Created by XMLmind XSL-FO Converter.

A separate semantic node is added for every record: `{'$gn', record, int()}`

- The only attribute for the node is `name`.
- Linked to
  - the record definition with `recdef` tag
  - the containing file with `record` tag
  - the expressions that refers to the record with `recref` tag
  - its fields with `field` tag

4.10. Record field information

A separate semantic node is added for every record field: `{'$gn', field, int()}`

- The only attribute for the node is `name`.
- Linked to
  - the record field definition with `fielddef` tag
  - the expressions that refers to the filed with `fieldref` tag

4.11. ETS table information

A separate semantic node is added for every ets table: `{'$gn', ets_tab, int()}`

- The only attribute for the node is `names`.
- Linked to
  - the expression that refers (`ets_ref`) or creates (`ets_def`) the ets table

4.12. PID information

A separate semantic node is added for every identified process identifier: `{'$gn', pid, int()}`

- The only attribute for the node is `reg_name`.
- Linked to
  - the expression that refers to the process

4.13. Environment information

A separate semantic node is added for every environmental information: `{'$gn', env, int()}`

- The attributes for the node are: `name`, `value`
- Linked to the root node

Created by XMLmind XSL-FO Converter.

Chapter 6. Lecture 6

1. Data-flow graph

1.1. Data-flow

- Gathering information about data handling and manipulation
- Possible sets of values at various points
- Different data-flow analyses:
  - constant-propagation
  - liveness analysis
  - available expression analysis
  - reaching definition analysis
  - etc.

1.2. Reaching definition analysis

- Erlang is a single assignment language, thus our interest is in reaching definition analysis
- Find those program points that can be a copy of a certain expression or variable
- The result of the analysis is a Data-Flow Graph (DFG)
- The DFG includes the direct and indirect relations among expressions
- DFG = (N , E)
  - $n_i \in N$ are nodes in the graph
  - $(n_i \rightarrow n_j) \in E$ are edges of the graph

1.3. Data-Flow analysis in RefactorErl

- Formal rules based on the syntax and semantics of the language
- Data-flow rules described with compositional syntax
- The rules are applied while traversing the SPG
- Applying the rules results in an interprocedural Data-Flow graph that is part of the SPG
- Indirect data flow/dependence can be calculated with transitive closure of the DFG edges
- Data-Flow Reaching

1.4. Kinds of Data-Flow edges

- $n_i \xrightarrow{f} n_j$ – the node n_j can be a copy of n_i
- $n_i \xrightarrow{c_k} n_j$ – the node n_j is a compound expression that contains the value of node n_i as its kth element

Created by XMLmind XSL-FO Converter.

- $n_i \xrightarrow{s_k} n_j$ – the node n_j is the kth element of the compound expression n_i
- $n_i \xrightarrow{d} n_j$ – the node n_j directly depends on the node n_i

1.5. Kinds of Data-Flow edges – Examples

- $n_i \xrightarrow{f} n_j$

  `A = 2`, $n_i = 2,\ n_j = A$
- $n_i \xrightarrow{c_k} n_j$

  `{1,2}`, $n_i = 1,\ n_j = \{1, 2\}$
- $n_i \xrightarrow{s_k} n_j$

  `{A,B}`, $n_i = \{A, B\},\ n_j = A$
- $n_i \xrightarrow{d} n_j$

  `{A + B}`, $n_j = A + B,\ n_i = A$

1.6. Notations used in the formal rules

- $e_i \in E$ – expressions
- $p_i \in E$ – pattern
- $m : f/n \in F$ – function f with arity n from module m
- $e'_0 \in E$ – special expression, that denotes the entire expression in case of a compound expression
- $fun(p_1, \ldots, p_n) - > \ldots$ – lambda function
- $funm : f/n$ – function expression for the given function

1.7. Data-flow rule: Variable

Expression: Edges: p binding of a variable

n occurrence of a variable $p \xrightarrow{f} n$

1.8. Variable – Example

Expression: Edges: $A = 2$

$B = 3$

$A\ \ 2 \xrightarrow{f} A$

$3 \xrightarrow{f} B$

$2 \xrightarrow{f} A$

Created by XMLmind XSL-FO Converter.

1.9. Data-flow rule: Match expression

Expression: Edges: e_0:

$p = e$ $e \xrightarrow{f} e_0$, $e \xrightarrow{f} p$

1.10. Match Expression – Example

Expression: Edges: $A = 2$

$2 \xrightarrow{f} A$

$2 \xrightarrow{f} =$

1.11. Data-flow rule: Pattern

Expression: Edges: p_0:

$p_1 = p_2$ $p_0 \xrightarrow{f} p_1$

$p_0 \xrightarrow{f} p_2$

1.12. Data-flow rule: Unary operator

Expression: Edges: e_0:

$\circ\, e_1$ $e_1 \xrightarrow{d} e_0$

1.13. Data-flow rule: Infix operator

Expression: Edges: e_0:

$e_1 \circ e_2$ $e_1 \xrightarrow{d} e_0$

$e_2 \xrightarrow{d} e_0$

1.14. Infix operator – Example

Expression: Edges: e_0:

$A + B$ $A \xrightarrow{d} +$

$B \xrightarrow{d} +$

1.15. Data-flow rule: Parenthesis

Expression: Edges: e_0:

(e) $e \xrightarrow{f} e_0$

1.16. Data-flow rule: Tuple expression

Expression: Edges: e_0:

$\{e_1, \ldots, e_n\}$ $e_1 \xrightarrow{c_1} e_0, \ldots, e_n \xrightarrow{c_n} e_0$

Created by XMLmind XSL-FO Converter.

1.17. Tuple expression – Example

Expression: Edges: e_0:

$\{1,2,3\}$ $1 \overset{c_1}{\to} \{1,2,3\}$

$2 \overset{c_2}{\to} \{1,2,3\}$

$3 \overset{c_3}{\to} \{1,2,3\}$

1.18. Data-flow rule: Tuple pattern

Expression: Edges: p_0:

$\{p_1,\ldots,p_n\}$ $p_0 \overset{s_1}{\to} p_1,\ldots,p_0 \overset{s_n}{\to} p_n$

1.19. Tuple pattern – Example

Expression: Edges: p_0:

$\{A,B,C\}$ $A \overset{s_1}{\to} \{A,B,C\}$

$B \overset{s_2}{\to} \{A,B,C\}$

$C \overset{s_3}{\to} \{A,B,C\}$

1.20. Data-flow rule: List expression

Expression: Edges: e_0:

$[e_1,\ldots,e_n|e_{n+1}]$ $e_1 \overset{c_\varsigma}{\to} e_0,\ldots,e_n \overset{c_\varsigma}{\to} e_0$

$e_{n+1} \overset{f}{\to} e_0$

1.21. Data-flow rule: List comprehension

Expression: Edges: e_0:

$[e_1||p \leftarrow e_2]$ $e_1 \overset{c_\varsigma}{\to} e_0, e_2 \overset{s_\varsigma}{\to} p$

1.22. Data-flow rule: List pattern

Expression: Edges: p_0:

$[p_1,\ldots,p_n|p_{n+1}]$ $p_0 \overset{s_\varsigma}{\to} p_1,\ldots,p_0 \overset{s_\varsigma}{\to} p_n$

$p_0 \overset{f}{\to} p_{n+1}$

1.23. Data-flow rule: BIF 1

Expression: Edges: e_0:

$\mathsf{hd}(e_1)$ $e_1 \overset{s_\varsigma}{\to} e_0$

1.24. BIF 1 – Example

Created by XMLmind XSL-FO Converter.

Expression: Edges: $\mathsf{hd}([1,2,3])$ $[1,2,3] \xrightarrow{s_\varsigma} hd([1,2,3])$

1.25. Data-flow rule: BIF 2

Expression: Edges: e_0:

$\mathsf{tl}(e_1)$ $e_1 \xrightarrow{f} e_0$

1.26. Data-flow rule: BIF 3

Expression: Edges: I constant,

e_0:

$\mathsf{element}(I, e_1)$ $e_1 \xrightarrow{s_I} e_0$

1.27. Data-flow rule: Case expression

Expression: Edges: e_0:

$\mathsf{case}\ e\ \mathsf{of}$

$p_1 when\ g_1 \rightarrow e_1^1, \ldots, e_{l_1}^1;$

$\vdots$

$p_n when\ g_n \rightarrow e_1^n, \ldots, e_{l_n}^n$

end $e \xrightarrow{f} p_1, \ldots, e \xrightarrow{f} p_n$

$e_{l_1}^1 \xrightarrow{f} e_0, \ldots, e_{l_n}^n \xrightarrow{f} e_0$

1.28. Case expression – Example

Expression: Edges: e_0:

$\mathsf{case}\ f()\ \mathsf{of}$

$[H|T] \rightarrow T, ok;$

$[] \rightarrow [], nok$

end $f() \xrightarrow{f} [H|T]$,

$f() \xrightarrow{f} []$

$ok \xrightarrow{f} case...end$,

$nok \xrightarrow{f} case...end$

1.29. Data-flow rule: If expression

Expression: Edges: e_0:

if

Created by XMLmind XSL-FO Converter.

$g_1 \to e^1_1, \ldots, e^1_{l_1};$

$\vdots$

$g_k \to e^k_1, \ldots, e^k_{l_k}$

end $e^1_{l_1} \xrightarrow{f} e_0, \ldots, e^n_{l_k} \xrightarrow{f} e_0$

1.30. Data-flow rule: Function call

Expression: Edges: e_0:

$\mathsf{m} : \mathsf{g}(e_1, \ldots, e_n)$ or

$\mathsf{g}(e_1, \ldots, e_n)$

m:g/n:

$\mathsf{g}(p^1_1, \ldots, p^1_n)\ when\ g_1 \ \to$

$e^1_1, \ldots, e^1_{l_1};$

$\vdots$

$\mathsf{g}(p^m_1, \ldots, p^m_n)\ when\ g_m \ \to$

$e^m_1, \ldots, e^m_{l_m}.\ e_1 \xrightarrow{f} p^1_1, \ldots, e_1 \xrightarrow{f} p^m_1$

$\vdots$

$e_n \xrightarrow{f} p^1_n, \ldots, e_n \xrightarrow{f} p^m_n$

$e^1_{l_1} \xrightarrow{f} e_0, \ldots, e^m_{l_m} \xrightarrow{f} e_0$

1.31. Function call – Example

Expression: Edges: $\mathsf{mymod} : \mathsf{myfun}([1,2,3], 0)$ $\mathsf{myfun}([H|T], Acc)) \ \to$

$NewAcc = H + Acc$,

$myfun(T, NewAcc);$

$\mathsf{myfun}([], Acc0) \ \to$

$Acc0.$ $[1,2,3] \xrightarrow{f} [H|T]$,

$[1,2,3] \xrightarrow{f} []$,

$0 \xrightarrow{f} Acc$,

$0 \xrightarrow{f} Acc0$

$myfun(T, NewAcc) \xrightarrow{f} mymod : myfun([1,2,3], 0)$,

Created by XMLmind XSL-FO Converter.

$Acc0 \xrightarrow{f} mymod : myfun([1, 2, 3], 0)$

1.32. Data-flow rule: Function call 2

Expression: Edges: e_0:

$e_m : e_g(e_1, \ldots, e_n)$

e_m or e_g is not constant or

$e_m : e_g/n$ is not defined

$e_m \xrightarrow{d} e_0$, $e_g \xrightarrow{d} e_0$

$e_1 \xrightarrow{d} e_0, \ldots, e_n \xrightarrow{d} e_0$

1.33. Data-flow rule: Try expression

Expression: Edges: e_0:

try $e_1, \ldots, e_k$of

$p_1\ when\ g_1 \rightarrow\ e_1^1, \ldots, e_{l_1}^1;$

$\vdots$

$p_n\ when\ g_n \rightarrow\ e_1^n, \ldots, e_{l_n}^n$

catch

$p_{n+1}\ when\ g_{n+1} \rightarrow$

$e_1^{n+1}, \ldots, e_{l_{n+1}}^{n+1};$

$\vdots$

$p_m\ when\ g_m \rightarrow\ e_1^m, \ldots, e_{l_m}^m$

after

$e_{m+1} \rightarrow\ e_1^{m+1}, \ldots, e_{l_{m+1}}^{m+1}$

end $e \xrightarrow{f} p_1, \ldots, e \xrightarrow{f} p_n$

$e_{l_1}^1 \xrightarrow{f} e_0, \ldots, e_{l_n}^n \xrightarrow{f} e_0$

$e_{l_{n+1}}^{n+1} \xrightarrow{f} e_0, \ldots, e_{l_m}^m \xrightarrow{f} e_0$

1.34. Data-flow rule: Catch expression

Expression: Edges: e_0:

catch e $e \xrightarrow{f} e_0$

Created by XMLmind XSL-FO Converter.

1.35. Data-flow rule: Block expression

Expression: Edges: e_0:

begin

$e_1, \ldots, e_n$

end $e_n \xrightarrow{f} e_0$

1.36. Data-flow rule: Send and receive expression

Expression: Edges: e_0:

$e_1 \; ! \; e_2$

e':

receive

$p_1 \; when \; g_1 \rightarrow$

$e_1^1, \ldots, e_{l_1}^1;$

$\vdots$

$p_n \; when \; g_n \rightarrow$

$e_1^n, \ldots, e_{l_n}^n$

$mathsfafter$

$e \rightarrow e_1, \ldots, e_s$

end $e_2 \xrightarrow{f} e_0$

$e_2 \xrightarrow{f} p_1, \ldots, e_2 \xrightarrow{f} p_n$

$e_{l_1}^1 \xrightarrow{f} e', \ldots, e_{l_n}^n \xrightarrow{f} e'$

$e_s \xrightarrow{f} e'$

1.37. Data-flow rule: Fun expression 1

Expression: Edges: e :

fun$(p_1^1, \ldots, p_n^1) \; when \; g_1 \; \rightarrow$

$e_1^1, \ldots, e_{l_1}^1;$

$\vdots$

$(p_1^m, \ldots, p_n^m) \; when \; g_m \; \rightarrow$

$e_1^m, \ldots, e_{l_m}^m$

Created by XMLmind XSL-FO Converter.

end

e_0:

$e(e_1, \ldots, e_n)$

e can be calculated

by data-flow analysis $e_1 \xrightarrow{f} p_1^1, \ldots, e_1 \xrightarrow{f} p_1^m$

$\vdots$

$e_n \xrightarrow{f} p_n^1, \ldots, e_n \xrightarrow{f} p_n^m$

$e_{l_1}^1 \xrightarrow{f} e_0, \ldots, e_{l_m}^m \xrightarrow{f} e_0$

1.38. Data-flow rule: Fun expression 2

Expression: Edges: *m:g/n*:

$\mathsf{g}(p_1^1, \ldots, p_n^1)\ when\ g_1\ \rightarrow$

$e_1^1, \ldots, e_{l_1}^1;$

$\vdots$

$\mathsf{g}(p_1^m, \ldots, p_n^m)\ when\ g_m\ \rightarrow$

$e_1^m, \ldots, e_{l_m}^m.$

e:

$\mathsf{fun}\ m : g/n\ or\ \mathsf{fun}\ g/n$

e_0:

$e(e_1, \ldots, e_n)$

e can be calculated

by data-flow analysis $e_1 \xrightarrow{f} p_1^1, \ldots, e_1 \xrightarrow{f} p_1^m$

$\vdots$

$e_n \xrightarrow{f} p_n^1, \ldots, e_n \xrightarrow{f} p_n^m$

$e_{l_1}^1 \xrightarrow{f} e_0, \ldots, e_{l_m}^m \xrightarrow{f} e_0$

1.39. Data-flow rule: Fun expression 3

Expression: Edges: e_0:

$e(e_1, \ldots, e_n)$

e can not be detected by data-flow reaching

Created by XMLmind XSL-FO Converter.

or e is m:g/n or g/n

by data-flow reaching

but m:g/n or g/n is not defined $e \xrightarrow{d} e_0$

$e_1 \xrightarrow{d} e_0, \ldots, e_n \xrightarrow{d} e_0$

1.40. Data-flow rule: Dynamic function call 1

Expression: Edges: e_0:

$e_1 : e_2(e_3, \ldots, e_{n+2})$

$e_1 : e_2/n$ is m:g/n

by data-flow reaching

m:g/n:

$\mathsf{g}(p_1^1, \ldots, p_n^1)\ when\ g_1\ \rightarrow$

$e_1^1, \ldots, e_{l_1}^1;$

$\vdots$

$\mathsf{g}(p_1^m, \ldots, p_n^m)\ when\ g_m\ \rightarrow$

$e_1^m, \ldots, e_{l_m}^m.\ e_3 \xrightarrow{f} p_1^1, \ldots, e_3 \xrightarrow{f} p_1^m$

$\vdots$

$e_{n+2} \xrightarrow{f} p_n^1, \ldots, e_{n+2} \xrightarrow{f} p_n^m$

$e_{l_1}^1 \xrightarrow{f} e_0, \ldots, e_{l_m}^m \xrightarrow{f} e_0$

1.41. Data-flow rule: Dynamic function call 2

Expression: Edges: e_0:

$\mathsf{apply}(e_1, e_2, e_3)$

e_1 is m, e_2 is g,

e_3 is $[e_4, \ldots e_{n+3}]$

determined by data-flow reaching

m:g/n:

$\mathsf{g}(p_1^1, \ldots, p_n^1)\ when\ g_1\ \rightarrow$

$e_1^1, \ldots, e_{l_1}^1;$

$\vdots$

Created by XMLmind XSL-FO Converter.

$\mathsf{g}(p_1^m, \ldots, p_n^m)\ when\ g_m\ \rightarrow$

$e_1^m, \ldots, e_{l_m}^m.\ e_4 \xrightarrow{f} p_1^1, \ldots, e_4 \xrightarrow{f} p_1^m$

$\vdots$

$e_{n+3} \xrightarrow{f} p_n^1, \ldots, e_{n+3} \xrightarrow{f} p_n^m$

$e_{l_1}^1 \xrightarrow{f} e_0, \ldots, e_{l_m}^m \xrightarrow{f} e_0$

1.42. Data-flow rule: Dynamic function call 3

Expression: Edges: e_0:

$e_1\ : e_2(e_3, \ldots, e_{n+2})$

$e_1, \ldots e_{n+2}$ cannot be

detected by data-flow reaching $e_1 \xrightarrow{d} e_0,\ e_2 \xrightarrow{d} e_0,\ e_3 \xrightarrow{d} e_0$

1.43. Data-flow rule: Dynamic function call 4

Expression: Edges: e_0:

$\mathsf{apply}(e_1, e_2, e_3)$

e_1, e_2, e_3 cannot be

detected by data-flow reaching $e_1 \xrightarrow{d} e_0,\ e_2 \xrightarrow{d} e_0,\ e_3 \xrightarrow{d} e_0$

Created by XMLmind XSL-FO Converter.

Chapter 7. Lecture 7

1. Data-flow graph

1.1. Data-Flow reaching

- Indirect data-flow can be computed from the data-flow graph by calculating the transitive closure of the graph
- The transitive closure is refined with a reaching relation
- Levels of the analysis:
  - 0th order analysis
  - 1st order analysis

  $\vdots$

2. 0th order data-flow analysis

2.1. 0th order data-flow reaching

- Calculating indirect dependencies
- Defined by $\overset{0f}{\rightsquigarrow}$ relation
- The relation $n_1 \overset{0f}{\rightsquigarrow} n_2$ means, that the value represented by n_1 in the DFG can be a copy of the value n_2 (their values are equal)
- Based on the following rules we can formalise the data-flow reaching relation: **reflexive rule, transitive rule, f rule, c-s rule**

2.2. 0th order data-flow reaching rules (1)

- **reflexive rule** – $n_1 \overset{0f}{\rightsquigarrow} n_1$ always holds, because the value of an expression reaches itself
- **transitive rule** – If the value of an expression n_1 reaches n_2 and the value of n_2 reaches n_3, then the value of n_1 reaches n_3, their values are equal

2.3. 0th data-flow reaching rules (2)

- **f rule** – If there is a flow edge $\overset{f}{\rightarrow}$ between nodes n_1 and n_2, then the value of n_1 reaches n_2
- **c-s rule** – A compound data structure preserves the data in its elements. When we put an element n_1 into a data structure n_2 and the compound data reaches another node n_3 and we take out the element from the compound data to n_4, then the packed value n_1 reaches n_4

2.4. Definition of 0th order data-flow reaching

The zeroth order data-flow reaching relation ($\overset{0f}{\rightsquigarrow}$) is the minimal relation that satisfies the followings:

$$n \overset{0f}{\rightsquigarrow} n$$

Created by XMLmind XSL-FO Converter.

$$\frac{n_1 \xrightarrow{f} n_2}{n_1 \overset{0f}{\rightsquigarrow} n_2} \quad \text{(f rule)}$$

$$\frac{n_1 \xrightarrow{c_i} n_2,\ n_2 \overset{0f}{\rightsquigarrow} n_3,\ n_3 \xrightarrow{s_i} n_4}{n_1 \overset{0f}{\rightsquigarrow} n_4} \quad \text{(c-s rule)}$$

$$\frac{n_1 \overset{0f}{\rightsquigarrow} n_2,\ n_2 \overset{0f}{\rightsquigarrow} n_3}{n_1 \overset{0f}{\rightsquigarrow} n_3} \quad \text{(transitive)}$$

2.5. Applications of 0th order data-flow reaching

- The last element of the flow chain is relevant for some applications
- We introduce forward and backward compact data-flow reaching
- Examples:
  - Dynamic function call analysis
  - Grokking

2.6. Definition of 0th order compact forward data-flow relation

The compact forward data-flow reaching ($\overset{0f_{cf}}{\rightsquigarrow}$) is the minimal relation that satisfies the following rule:

$$\frac{n_1 \overset{0f}{\rightsquigarrow} n_2,\ \nexists n_3,\ n_3 \neq n_2 :\ n_2 \overset{0f}{\rightsquigarrow} n_3}{n_1 \overset{0f_{cf}}{\rightsquigarrow} n_2} \quad \text{(f-compact)}$$

2.7. Definition of 0th order compact backward data-flow relation

The compact backward data-flow reaching ($\overset{0f_{cb}}{\rightsquigarrow}$) is the minimal relation that satisfies the following rules:

$$\frac{n_1 \overset{0f}{\rightsquigarrow} n_2,\ \nexists n_0,\ n_0 \neq n_1 :\ n_0 \overset{0f}{\rightsquigarrow} n_1}{n_1 \overset{0f_{cb}}{\rightsquigarrow} n_2} \quad \text{(b-compact)}$$

3. 0th order DFG and reaching example

3.1. Example module

```
  -module(dataflow).

swap({A, B}) ->
```

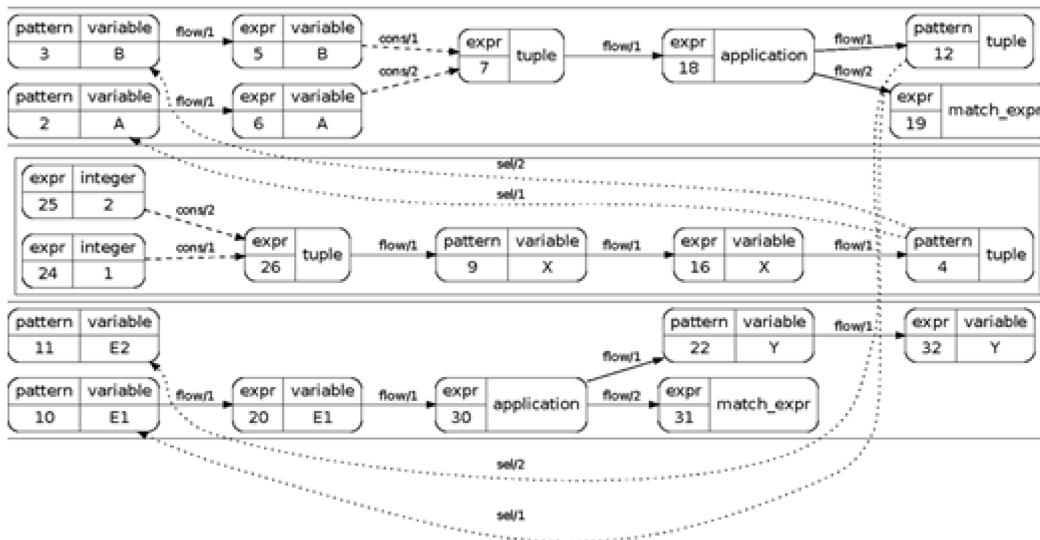

Created by XMLmind XSL-FO Converter.

```
        {B, A}.


    get_1st(X) ->
        {E1, E2} = swap(X),
        E1.

    const()->
        Y = get_1st({1,2}),
        Y.
```

3.2. DFG for `dataflow` module



3.3. Applying data-flow reaching rules (1)

Applying the `f rule` 4 times:

$$\frac{e_{22} \xrightarrow{f} e_{32},\ e_{30} \xrightarrow{f} e_{22},\ e_{20} \xrightarrow{f} e_{30},\ p_{10} \xrightarrow{f} e_{20}}{e_{22} \overset{0f}{\rightsquigarrow} e_{32},\ e_{30} \overset{0f}{\rightsquigarrow} e_{22},\ e_{20} \overset{0f}{\rightsquigarrow} e_{30},\ p_{10} \overset{0f}{\rightsquigarrow} e_{20}}$$

3.4. Applying data-flow reaching rules (2)

Applying the `transitive rule` 3 times:

$$\frac{p_{10} \overset{0f}{\rightsquigarrow} e_{20},\ e_{20} \overset{0f}{\rightsquigarrow} e_{30},\ e_{30} \overset{0f}{\rightsquigarrow} e_{22},\ e_{22} \overset{0f}{\rightsquigarrow} e_{32}}{p_{10} \overset{0f}{\rightsquigarrow} e_{32}}$$

3.5. Applying data-flow reaching rules (3)

Applying the `f rule` 2 times:

$$\frac{e_{18} \xrightarrow{f} e_{12},\ e_{7} \xrightarrow{f} e_{18}}{e_{18} \overset{0f}{\rightsquigarrow} e_{12},\ e_{7} \overset{0f}{\rightsquigarrow} e_{18}}$$

3.6. Applying data-flow reaching rules (4)

Created by XMLmind XSL-FO Converter.

Applying the `f rule`:

$$\frac{p_2 \stackrel{f}{\rightarrow} e_6}{p_2 \stackrel{0f}{\rightsquigarrow} e_6}$$

3.7. Applying data-flow reaching rules (5)

Applying the `f rule` 3 times:

$$\frac{e_{16} \stackrel{f}{\rightarrow} p_4,\ e_9 \stackrel{f}{\rightarrow} e_{16},\ e_{26} \stackrel{f}{\rightarrow} e_9}{e_{16} \stackrel{0f}{\rightsquigarrow} e_4,\ e_9 \stackrel{0f}{\rightsquigarrow} e_{16},\ e_{26} \stackrel{0f}{\rightsquigarrow} e_9}$$

3.8. Applying data-flow reaching rules (6)

Applying the `transitive rule` 2 times:

$$\frac{e_{29} \stackrel{f}{\rightarrow} e_9,\ e_9 \stackrel{f}{\rightarrow} e_{16},\ e_{16} \stackrel{f}{\rightarrow} p_4}{e_{26} \stackrel{0f}{\rightsquigarrow} p_4}$$

3.9. Applying data-flow reaching rules (7)

Applying the `c-s rule`:

$$\frac{e_6 \stackrel{c_1}{\rightarrow} e_7,\ e_7 \stackrel{0f}{\rightsquigarrow} e_{12},\ e_{12} \stackrel{s_1}{\rightarrow} p_{10}}{e_6 \stackrel{0f}{\rightsquigarrow} p_{10}} \qquad \text{(c-s rule)}$$

3.10. Applying data-flow reaching rules (8)

Applying the `c-s rule`:

$$\frac{e_{25} \stackrel{c_2}{\rightarrow} e_{26},\ e_{26} \stackrel{0f}{\rightsquigarrow} p_4,\ p_4 \stackrel{s_2}{\rightarrow} p_2}{e_{25} \stackrel{0f}{\rightsquigarrow} p_2}$$

3.11. Applying data-flow reaching rules (9)

Applying the `transitive rule` 3 times:

$$\frac{e_{25} \stackrel{0f}{\rightsquigarrow} p_2,\ p_2 \stackrel{0f}{\rightsquigarrow} e_6,\ e_6 \stackrel{0f}{\rightsquigarrow} p_{10},\ p_{10} \stackrel{0f}{\rightsquigarrow} e_{32}}{e_{25} \stackrel{0f}{\rightsquigarrow} e_{32}}$$

Created by XMLmind XSL-FO Converter.

Chapter 8. Lecture 8

1. Control-Flow

1.1. Control-Flow Analysis

- Technique for determining the control flow of a program
- Control-Flow Graph - CFG
- Every execution path
- The nodes of the CFG are the expressions
- An edge represents direct control-flow relation between two nodes

1.2. Control-Flow Graph in general

- $CFG = (N, V, l)$ is a directed graph
- N is a set of node identifiers
- $V \subseteq N \times N$ is a set of edges between the nodes
- $l : N \times N \longmapsto L$ is a function that assigns labels to edges
- $L := \{\epsilon, \texttt{yes}, \texttt{no}, \texttt{ret}, \texttt{call}, \texttt{rec}, \texttt{send}\}$

1.3. Control-Flow Graph for Erlang

- The nodes of the graph are the expressions of the SPG
- The relations/edges between the nodes are defined by right of formal rules
- There are edges with no labels and labeled edges in the rules
- The labeled edges have special role

1.4. Notations in rules

- $e, e_i \in E$ are expressions
- $g, g_j \in G$ are guard expressions
- $f/n \in F$ stands for function f with arity n
- $e'_0 \in E$ is a special expression, that denotes the entry point of the e_0 compound expression
- $\circ_k$ is an infix binary operator
- $e' \rightarrow e''$, $e' \xrightarrow{l} e''$ denote the control-flow from expression e' to e'', l is the label of the edge

1.5. Control-flow rule: Unary operator

Expression: Edges: e_0:

Created by XMLmind XSL-FO Converter.

$\circ \; e_1 \; e'_0 \rightarrow e_1, e_1 \rightarrow e_0$

1.6. Unary operator – Example

Expression: Edges: $\circ \; f() \; f() \rightarrow \circ$

1.7. Control-flow rule: Left associative operator

Expression: Edges: e_0:

$e_1 \; \circ_1 \; e_2 \; \circ_2 \ldots$

$\circ_{n-2} \; e_{n-1} \; \circ_{n-1} \; e_n \; e'_0 \rightarrow e_1,$

$e_1 \rightarrow e_2, e_2 \rightarrow \circ_1, \circ_1 \rightarrow e_3 \ldots$

$e_n \rightarrow \circ_{n-1}, \circ_{n-1} \rightarrow e_0$

1.8. Left associative operator – Example

Expression: Edges: $1 + 2 +' 3 \; 1 \rightarrow 2,$

$2 \rightarrow +,$

$+ \rightarrow 3,$

$3 \rightarrow +'$

1.9. Control-flow rule: Right associative operator

Expression: Edges: e_0:

$e_1 \; \circ_1 \; e_2 \circ_2 \ldots$

$\circ_{n-2} \; e_{n-1} \; \circ_{n-1} \; e_n \; e'_0 \rightarrow e_1,$

$e_1 \rightarrow e_2, \ldots, e_{n-1} \rightarrow e_n,$

$e_n \rightarrow \circ_{n-1},$

$\circ_{n-1} \rightarrow \circ_{n-2}, \ldots, \circ_2 \rightarrow \circ_1,$

$\circ_1 \rightarrow e_0$

1.10. Control-flow rule: Comparison operator

Expression: Edges: e_0:

$e_1 \circ e_2 \; e'_0 \rightarrow e_1,$

$e_1 \rightarrow e_2,$

$e_2 \rightarrow e_0$

1.11. Control-flow rule: Andalso operator

Expression: Edges: e_0:

Created by XMLmind XSL-FO Converter.

$e_1 \circ e_2$ $e'_0 \to e_1$,

$e_1 \stackrel{yes}{\to} e_2$,

$e_1 \stackrel{no}{\to} e_0$,

$e_2 \to e_0$

1.12. Andalso operator – Example

Expression: Edges: A andalso B $A \stackrel{yes}{\to} B$,

$A \stackrel{no}{\to} andalso$,

$B \to andalso$

1.13. Control-flow rule: Orelse operator

Expression: Edges: e_0:

$e_1 \circ e_2$ $e'_0 \to e_1$,

$e_1 \stackrel{no}{\to} e_2$,

$e_1 \stackrel{yes}{\to} e_0$,

$e_2 \to e_0$

1.14. Control-flow rule: Send operator

Expression: Edges: e_0:

$e_1 \; ! \; e_2$

$e'_0 \to e_1$,

$e_1 \to e_2$,

$e_2 \stackrel{send}{\to} e_0$

1.15. Control-flow rule: Parenthesis

Expression: Edges: e_0:

(e_1) $e'_0 \to e_1$,

$e_1 \to e_0$

1.16. Control-flow rule: Tuple expression

Expression: Edges: e_0:

$\{e_1, \ldots, e_n\}$ $e'_0 \to e_1$,

$e_1 \to e_2, \ldots, e_{n-1} \to e_n$,

Created by XMLmind XSL-FO Converter.

$e_n \to e_0$

1.17. Control-flow rule: List expression

Expression: Edges: e_0:

$[e_1, \ldots, e_n | e_{n+1}]$ $e_0' \to e_1$,

$e_1 \to e_2, \ldots, e_n \to e_{n+1}$,

$e_{n+1} \to e_0$

1.18. Control-flow rule: List comprehension (1)

Expression: Edges: e_0:

$[e || p_1 < -e_1, \ldots, p_n < -e_n]$

$e_0' \to e_1$,

$e_i \to p_i, p_i \stackrel{no}{\to} e_i, p_i \stackrel{yes}{\to} e_{i+1}$,

$e_i \stackrel{ret}{\to} e_{i-1}$,

$e \to e_1$,

$(i \in [1, ..., n], e_{n+1} = e)$

1.19. Control-flow rule: List comprehension (2)

Expression: Edges: e_0:

$[e || p_1 < -e_1, f_{(1,0)}, \ldots, f_{(1,m_1)}$

$\vdots$

$p_n < -e_n, f_{(n,0)}, \ldots, f_{(1,m_n)}]$ $e_0' \to e_1$,

$e_i \to p_i, p_i \stackrel{no}{\to} e_i$,

$e_i \stackrel{ret}{\to} e_{i-1}$,

$p_i \stackrel{yes}{\to} f_{(i,0)}$,

$f_{(i,j-1)} \stackrel{yes}{\to} f_{(i,j)}, f_{(i,m_i)} \stackrel{yes}{\to} e_{i+1}$,

$f_{(i,0)} \stackrel{no}{\to} e_i, f_{(i,j)} \stackrel{no}{\to} e_i$,

$e \to e_1$,

$(i \in [1, ..., n], j \in [1, ..., m_i]$,

$n, m_i \in N e_{n+1} = e)$

1.20. Control-flow rule: List comprehension (3)

Created by XMLmind XSL-FO Converter.

Expression: Edges: e_0:

$[e||f_{(0,0)},\ldots,f_{(0,m_0)}$,

$p_1 < -e_1,\ldots,p_n < -e_n]$

$e_i \rightarrow p_i, p_i \overset{no}{\rightarrow} e_i, p_i \overset{yes}{\rightarrow} e_{i+1},$

$e_i \overset{ret}{\rightarrow} e_{i-1}$,

$e'_0 \rightarrow f_{(0,0)}$,

$f_{(0,j-1)} \overset{yes}{\rightarrow} f_{(0,j)}, f_{(0,m_0)} \overset{yes}{\rightarrow} e_1$,

$f_{(0,0)} \overset{no}{\rightarrow} e_0, f_{(0,j)} \overset{no}{\rightarrow} e_0$,

$e \rightarrow e_1$,

$(i \in [1,...,n], j \in [1,...,m_0]$,

$n, m_0 \in N e_{n+1} = e)$

1.21. List comprehension – Example

Expression: Edges: $[X * 2||X < -List, X > 10]$

$List \rightarrow X$,

$X \overset{no}{\rightarrow} List, X \overset{yes}{\rightarrow} (X > 10)$,

$(X > 10) \overset{no}{\rightarrow} List$,

$(X > 10) \overset{yes}{\rightarrow} X * 2$,

$(X * 2) \rightarrow List$

1.22. Control-flow rule: Function application

Expression: Edges: e_0:

$e_f(e_1,\ldots,e_n)$

$e'_0 \rightarrow e_f,$

$e_f \rightarrow e_1,$

$e_1 \rightarrow e_2,\ldots,e_{n-1} \rightarrow e_n,$

$e_n \overset{call}{\rightarrow} e_0$

1.23. Function application – Example

Expression: Edges: $\mathsf{myfun}(1,2)$

Created by XMLmind XSL-FO Converter.

$myfun \to 1,$

$1 \to 2,$

$2 \stackrel{call}{\to} myfun(1,2)$

1.24. Function definition

f/n:

$f(p_1^1, \ldots, p_n^1)$ [when g_1] $\to e_1^1, \ldots, e_{l_1}^1;$

$\vdots$

$f(p_1^m, \ldots, p_n^m)$ [when g_n] $\to e_1^n, \ldots, e_{l_n}^n$

1.25. Control-flow rule: Function definition

$f/n \to p_1^1$

$\{p_1^1, \ldots, p_n^1\} \stackrel{yes}{\to} g^1$

$\{p_1^1, \ldots, p_n^1\} \stackrel{no}{\to} \{p_1^2, \ldots, p_n^2\}$

$\vdots$

$\{p_1^{m-1}, \ldots, p_n^{m-1}\} \stackrel{yes}{\to} g^{m-1}$

$\{p_1^{m-1}, \ldots, p_n^{m-1}\} \stackrel{no}{\to} \{p_1^m, \ldots, p_n^m\}$

$\{p_1^m, \ldots, p_n^m\} \stackrel{yes}{\to} g^m$

$\{p_1^m, \ldots, p_n^m\} \stackrel{no}{\to} error$

$g^1 \stackrel{yes}{\to} e_1^1$

$g^1 \stackrel{no}{\to} \{p_1^2, \ldots, p_n^2\}$

$\vdots$

$g^{m-1} \stackrel{yes}{\to} e_1^{m-1}$

$g^{m-1} \stackrel{no}{\to} \{p_1^m, \ldots, p_n^m\}$

$g^m \stackrel{yes}{\to} e_1^m$

$g^m \stackrel{no}{\to} error$

$e_1^1 \to e_2^1, \ \ldots, \ e_{l_1-1}^1 \to e_{l_1}^1$

$\vdots$

$e_1^m \to e_2^m, \ \ldots, \ e_{l_m-1}^m \to e_{l_m}^m$

$e_{l_1}^1 \to ret \ f/n$

$\vdots$

Created by XMLmind XSL-FO Converter.

$e^m_{l_m} \rightarrow ret\ f/n$

1.26. Function – Example

$\mathsf{myfun}(N)$ when $N > 0 \ \rightarrow$

$Temp = \mathsf{myfun}(N-1),$

$Temp * N;$

$\mathsf{myfun}(0) \ \rightarrow$

1. $myfun/1 \rightarrow \{N\}$,

$\{N\} \stackrel{yes}{\rightarrow} N > 0,$

$\{N\} \stackrel{no}{\rightarrow} \{0\},$

$\{0\} \stackrel{yes}{\rightarrow} 1,$

$\{0\} \stackrel{no}{\rightarrow} error,$

$N > 0 \stackrel{yes}{\rightarrow} Temp = myfun(N-1),$

$N > 0 \stackrel{no}{\rightarrow} \{0\},$

$Temp = myfun(N-1) \stackrel{no}{\rightarrow} Temp * N,$

$Temp * N \rightarrow ret\ myfun/1$

$1 \rightarrow ret\ myfun/1$

1.27. Case expression

e_0:

$case\ e\ \ of$

p_1 [when g_1] $\rightarrow$

$e^1_1, \ldots, e^1_{l_1};$

$\vdots$

p_n [when g_n] $\rightarrow$

$e^n_1, \ldots, e^n_{l_n};$

end

1.28. Control-flow rule: Case expression

$e'_0 \stackrel{e}{\rightarrow},\ e \rightarrow p_1,$

$p_1 \stackrel{yes}{\rightarrow} g_1,\ p_1 \stackrel{no}{\rightarrow} p_2,$

Created by XMLmind XSL-FO Converter.

$\vdots$

$p_{n_1} \stackrel{yes}{\rightarrow} g_{n-1},\ p_{n-1} \stackrel{no}{\rightarrow} p_n,$

$p_n \stackrel{yes}{\rightarrow} g_n,\ p_n \stackrel{no}{\rightarrow} error,$

$g_1 \stackrel{yes}{\rightarrow} e_1^1,\ g_1 \stackrel{no}{\rightarrow} p_2,$

$\vdots$

$g_{n-1} \stackrel{yes}{\rightarrow} e_1^{n-1},\ g_{n-1} \stackrel{no}{\rightarrow} p_n,$

$g_n \stackrel{yes}{\rightarrow} e_1^n,\ g_n \stackrel{no}{\rightarrow} error,\ e_1^1 \rightarrow e_2^1, \ldots, e_{l_1-1}^1 \rightarrow e_{l_1}^1,$

$\vdots$

$e_1^n \rightarrow e_2^n, \ldots, e_{l_n-1}^n \rightarrow e_{l_n}^n,$

$e_{l_1}^1 \rightarrow retcase,$

$\vdots$

$e_{l_n}^n \rightarrow retcase,$

$retcase \rightarrow e_0$

1.29. Receive expression

e_0:

$receive$

p_1 [when g_1] $\longrightarrow$

$e_1^1, \ldots, e_{l_1}^1;$

$\vdots$

p_n [when g_n] $\longrightarrow$

$e_1^n, \ldots, e_{l_n}^n;$

end

1.30. Control-flow rule: Receive expression

$e_0' \stackrel{e}{\rightarrow},\ e \stackrel{rec}{\rightarrow} p_1,$

$p_1 \stackrel{yes}{\rightarrow} g_1,\ p_1 \stackrel{no}{\rightarrow} p_2,$

$\vdots$

$p_{n_1} \stackrel{yes}{\rightarrow} g_{n-1},\ p_{n-1} \stackrel{no}{\rightarrow} p_n,$

$p_n \stackrel{yes}{\rightarrow} g_n,$

Created by XMLmind XSL-FO Converter.

$g_1 \overset{yes}{\to} e_1^1,\ g_1 \overset{no}{\to} p_2,$

$\vdots$

$g_{n-1} \overset{yes}{\to} e_1^{n-1},\ g_{n-1} \overset{no}{\to} p_n,$

$g_n \overset{yes}{\to} e_1^n,\ e_1^1 \to e_2^1, \ldots, e_{l_1-1}^1 \to e_{l_1}^1,$

$\vdots$

$e_1^n \to e_2^n, \ldots, e_{l_n-1}^n \to e_{l_n}^n,$

$e_{l_1}^1 \to retreceive,$

$\vdots$

$e_{l_n}^n \to retreceive,$

$retreceive \to e_0$

Created by XMLmind XSL-FO Converter.

Chapter 9. Lecture 9

1. Dominators and Postdominators

1.1. Dominators

- Node `n` dominates node `m` (`n dom m`), if every execution path from entry to `m` includes `n`
- The `dom` relation is:
  - *Reflexive* : every node dominates itself
  - *Transitive* : if `a dom b` and `b dom c`, then `a dom c`
  - *Antisymmetric* : if `a dom b` and `b dom a`, then `a = b`
- Immediate dominance relation (idom)

1.2. Immediate Dominance

- Subrelation of the dominance
- For `a` $\neq$ `b`, `a idom b` if and only if there does not exist a node `c` such that `c` $\neq$ `a` and `c` $\neq$ `b` for which `a dom c` and `c dom b`
- The immediate dominator is unique
- The immediate dominance relation forms a tree

1.3. Postdominators

- Postdominator relation: `b pdom a`, if every execution path from `a` to `exit` includes `b`
- Immediate postdominator: `b ipdom a`, if and only if `b pdom a` and does not exist a node `c` such that `a` $\neq$ `c` and `b` $\neq$ `c` for which `c pdom a` and `b pdom c` and `a` $\neq$ `b`
- Similar as dominator relation (reversed edges and `entry` and `exit` interchanged)

1.4. Postdominator calculating algorithm

Notations used in the algorithm:

- N – the set of nodes in the CFG
- e – the dummy exit node ($e \in N$)
- $pdom(n)$ – the set of postdominators of $n \in N$
- $ipdom(n)$ – the set of immediate postdominator (the size of the set is either zero or one)
- $succ(n)$ – the set of successors of $n \in N$

1.5. Postdominator calculating algorithm (cont.)

Initializing the postdominator sets:

Created by XMLmind XSL-FO Converter.

- $\forall n \in N \backslash \{e\} : pdom(n) := N$
- $pdom(e) := \{e\}$

Iterate the following step until there is no change in the postdominator set:

- $\forall n \in N \backslash \{e\} : pdom(n) := (\bigcap_{s \in succ(n)} pdom(s)) \cup \{n\}$

1.6. Calculating Immediate Postdominator

From the previously obtained postdominator sets we determine the immediate postdominators for each $n \in N$

- Initializing the immediate dominator sets:
  - $\forall n \in N : ipdom(n) := pdom(n) \backslash \{n\}$
- The main step of the calculation:
  - $\forall n \in N \backslash \{e\}, \forall s, t \in ipdom(n), s \neq t :$
    $ipdom(n) := \begin{cases} ipdom(n) \backslash \{t\} & \text{, if } t \in ipdom(s); \\ ipdom(n) & \text{, otherwise.} \end{cases}$

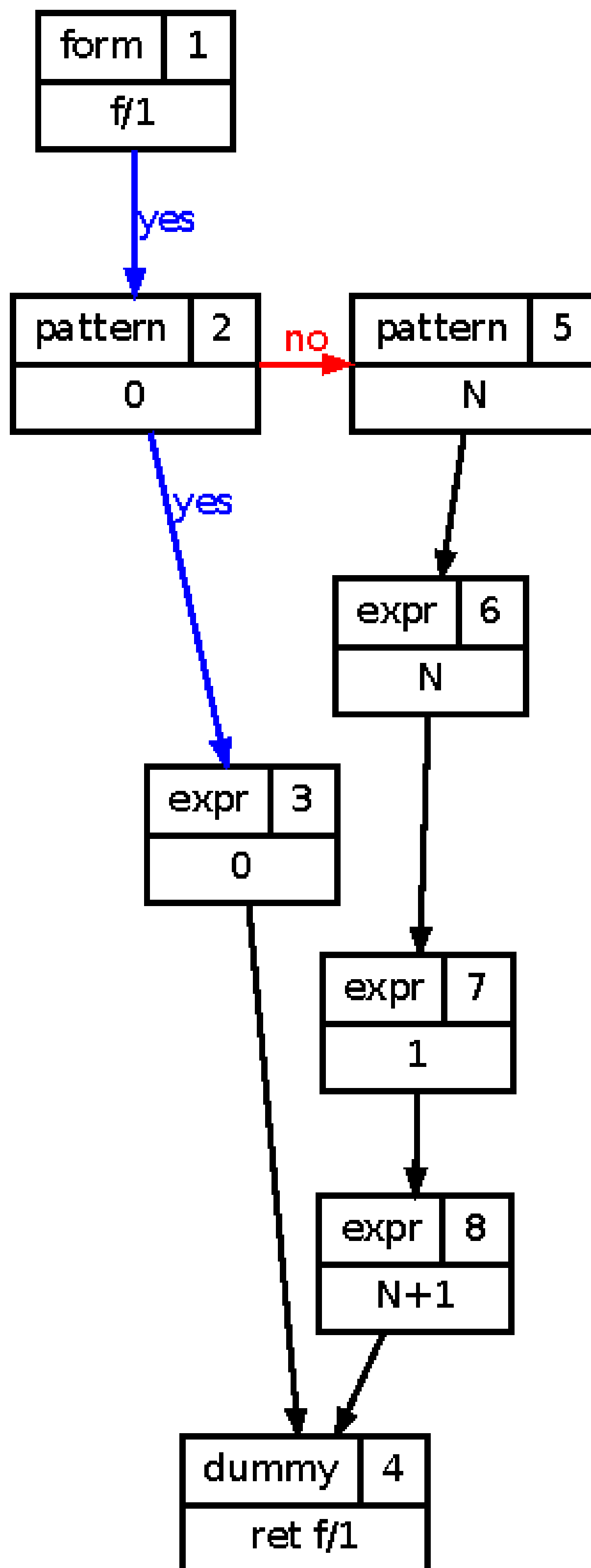
1.7. Example: CFG

- Function:

```
        f(0) -> 0;
f(N) -> N+1.
```

- The node `1` is the entry point of the function
- The node `4` is the return point of the function

Created by XMLmind XSL-FO Converter.



Created by XMLmind XSL-FO Converter.

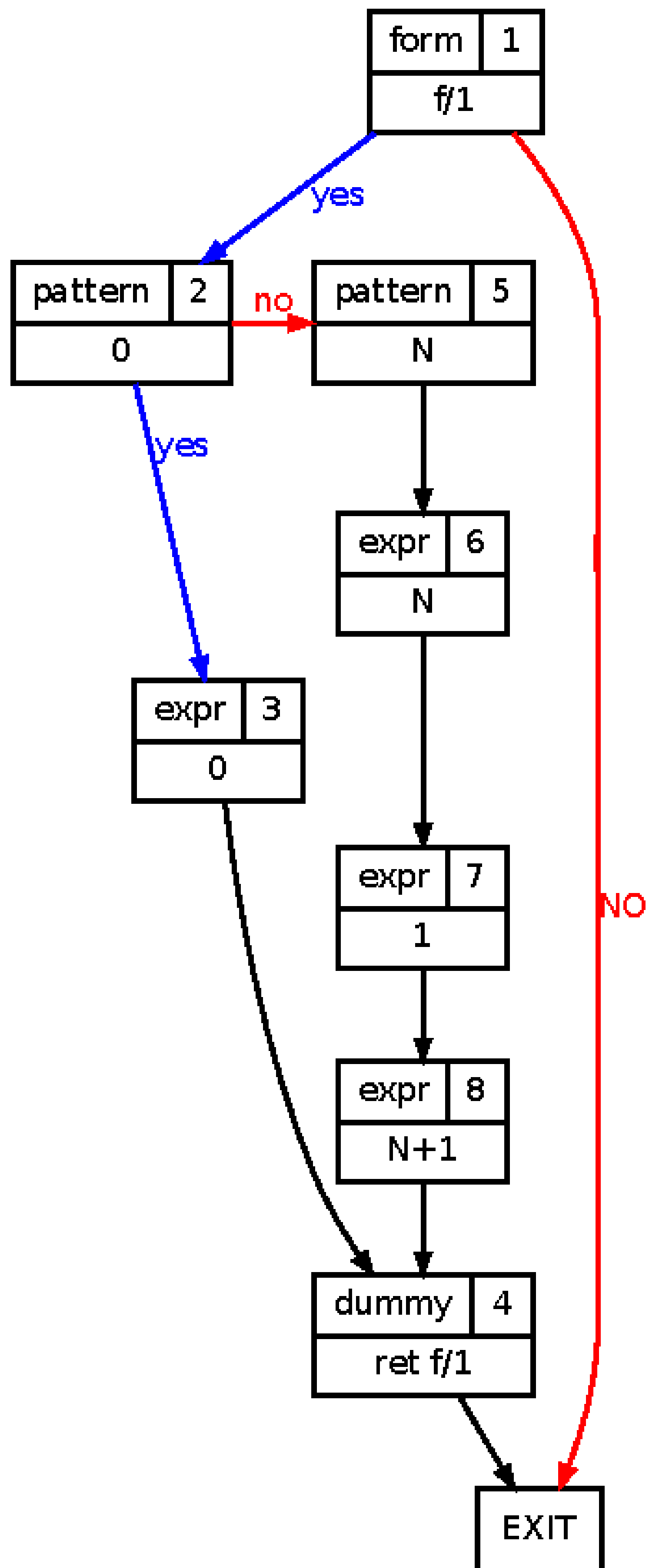
1.8. Example: CFG

- Function:

```
        f(0) -> 0;
f(N) -> N+1.
```

- The CFG is extended with a special node `EXIT`
- Two extra edges are also added, $1 \stackrel{NO}{\rightarrow} EXIT$ and $4 \rightarrow EXIT$

Created by XMLmind XSL-FO Converter.


form
1
f/1
yes
NO
pattern
2
0
no
pattern
5
N
yes
expr
6
N
expr
3
0
expr
7
1
expr
8
N+1
dummy
4
ret f/1
EXIT

1.9. Example: Postdominators

Initialised sets:

1 : { 1,2,3,4,5,6,7,8,EXIT }

2 : { 1,2,3,4,5,6,7,8,EXIT }

3 : { 1,2,3,4,5,6,7,8,EXIT }

4 : { 1,2,3,4,5,6,7,8,EXIT }

5 : { 1,2,3,4,5,6,7,8,EXIT }

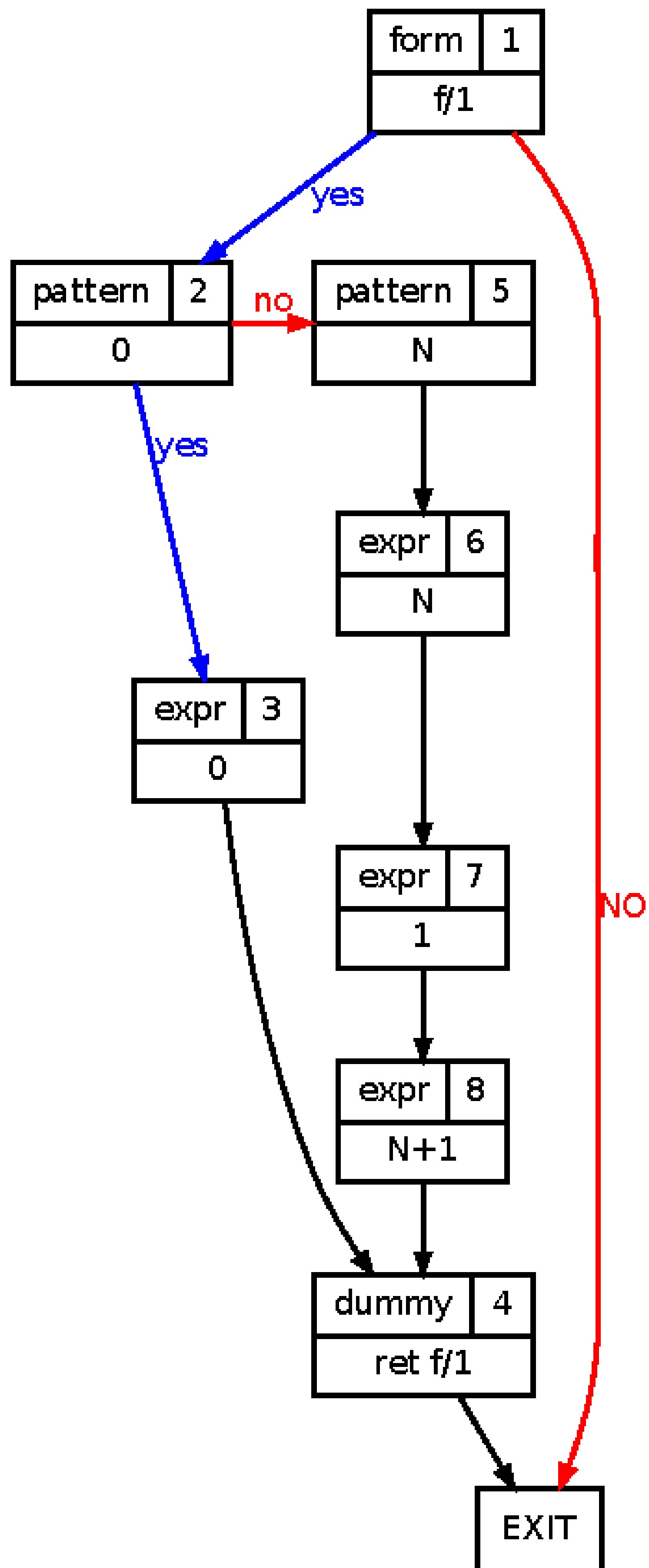
6 : { 1,2,3,4,5,6,7,8,EXIT }

7 : { 1,2,3,4,5,6,7,8,EXIT }

8 : { 1,2,3,4,5,6,7,8,EXIT }

EXIT : { EXIT }

Created by XMLmind XSL-FO Converter.


form
1
f/1
yes
pattern
2
0
no
pattern
5
N
yes
expr
6
N
expr
3
0
expr
7
1
NO
expr
8
N+1
dummy
4
ret f/1
EXIT

1.10. Example: Postdominators

Initialised sets:

1 : { 1,2,3,4,5,6,7,8,EXIT }

2 : { 1,2,3,4,5,6,7,8,EXIT }

3 : { 1,2,3,4,5,6,7,8,EXIT }

4 : { 1,2,3,4,5,6,7,8,EXIT }

5 : { 1,2,3,4,5,6,7,8,EXIT }

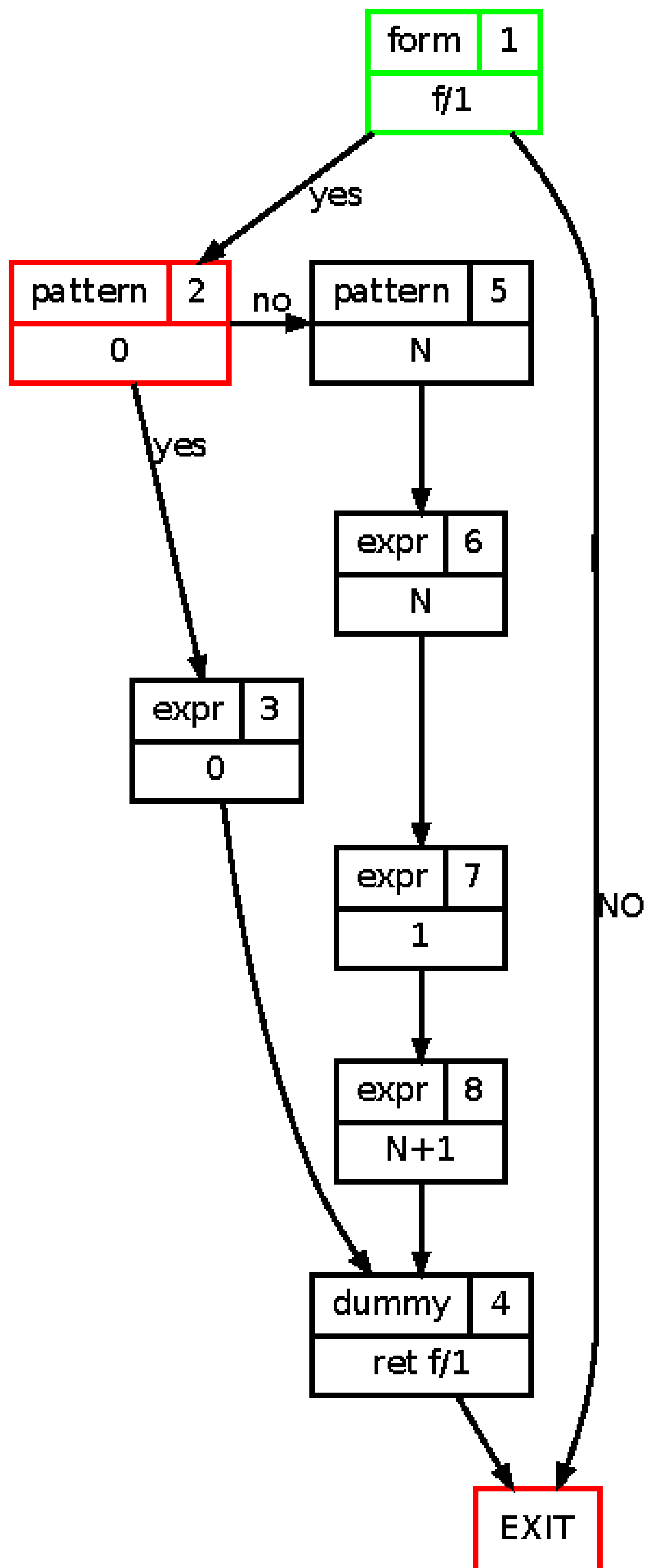
6 : { 1,2,3,4,5,6,7,8,EXIT }

7 : { 1,2,3,4,5,6,7,8,EXIT }

8 : { 1,2,3,4,5,6,7,8,EXIT }

EXIT : { EXIT }

Created by XMLmind XSL-FO Converter.


form
1
f/1
yes
NO
pattern
2
0
no
pattern
5
N
yes
expr
6
N
expr
3
0
expr
7
1
expr
8
N+1
dummy
4
ret f/1
EXIT

1.11. Example: Postdominators

Postdominator sets:

1 : { EXIT } $\cup \{1\}$

2 : { 1,2,3,4,5,6,7,8,EXIT }

3 : { 1,2,3,4,5,6,7,8,EXIT }

4 : { 1,2,3,4,5,6,7,8,EXIT }

5 : { 1,2,3,4,5,6,7,8,EXIT }

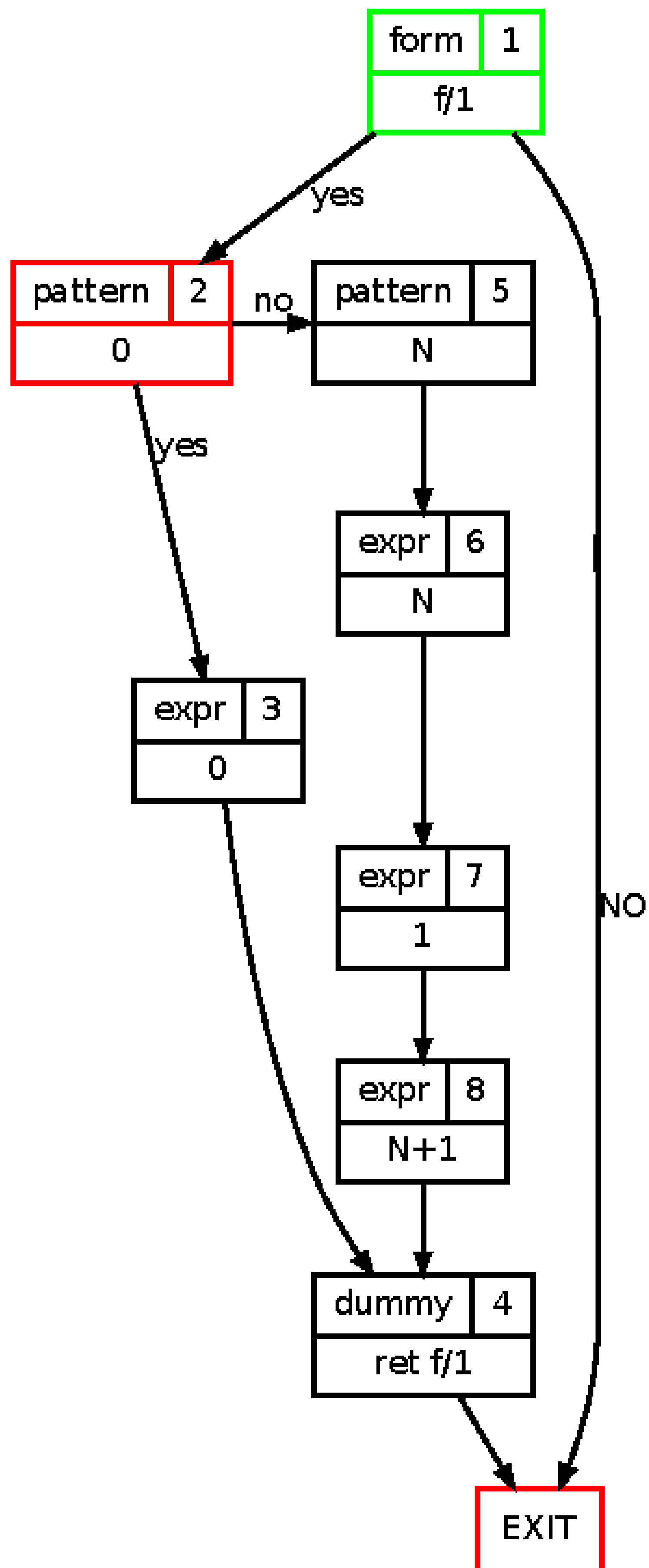
6 : { 1,2,3,4,5,6,7,8,EXIT }

7 : { 1,2,3,4,5,6,7,8,EXIT }

8 : { 1,2,3,4,5,6,7,8,EXIT }

EXIT : { EXIT }

Created by XMLmind XSL-FO Converter.


form
1
f/1
yes
pattern
2
0
no
pattern
5
N
yes
expr
6
N
expr
3
0
expr
7
1
NO
expr
8
N+1
dummy
4
ret f/1
EXIT

1.12. Example: Postdominators

Postdominator sets:

1 : { 1,EXIT }

2 : { 1,2,3,4,5,6,7,8,EXIT }

3 : { 1,2,3,4,5,6,7,8,EXIT }

4 : { 1,2,3,4,5,6,7,8,EXIT }

5 : { 1,2,3,4,5,6,7,8,EXIT }

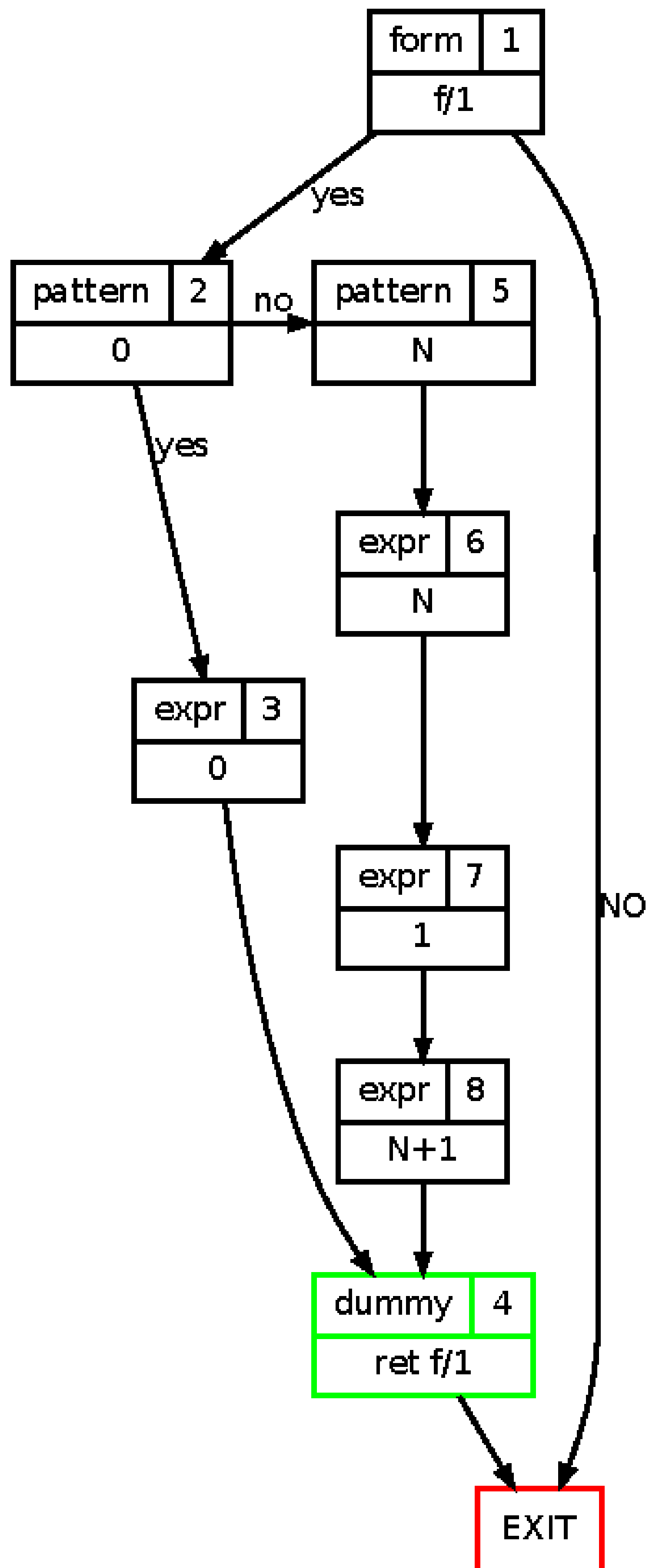
6 : { 1,2,3,4,5,6,7,8,EXIT }

7 : { 1,2,3,4,5,6,7,8,EXIT }

8 : { 1,2,3,4,5,6,7,8,EXIT }

EXIT : { EXIT }

Created by XMLmind XSL-FO Converter.


form
1
f/1
yes
NO
pattern
2
0
no
pattern
5
N
yes
expr
6
N
expr
3
0
expr
7
1
expr
8
N+1
dummy
4
ret f/1
EXIT

1.13. Example: Postdominators

Postdominator sets:

1 : { 1,EXIT }

2 : { 1,2,3,4,5,6,7,8,EXIT }

3 : { 1,2,3,4,5,6,7,8,EXIT }

4 : { EXIT } $\cup \{4\}$

5 : { 1,2,3,4,5,6,7,8,EXIT }

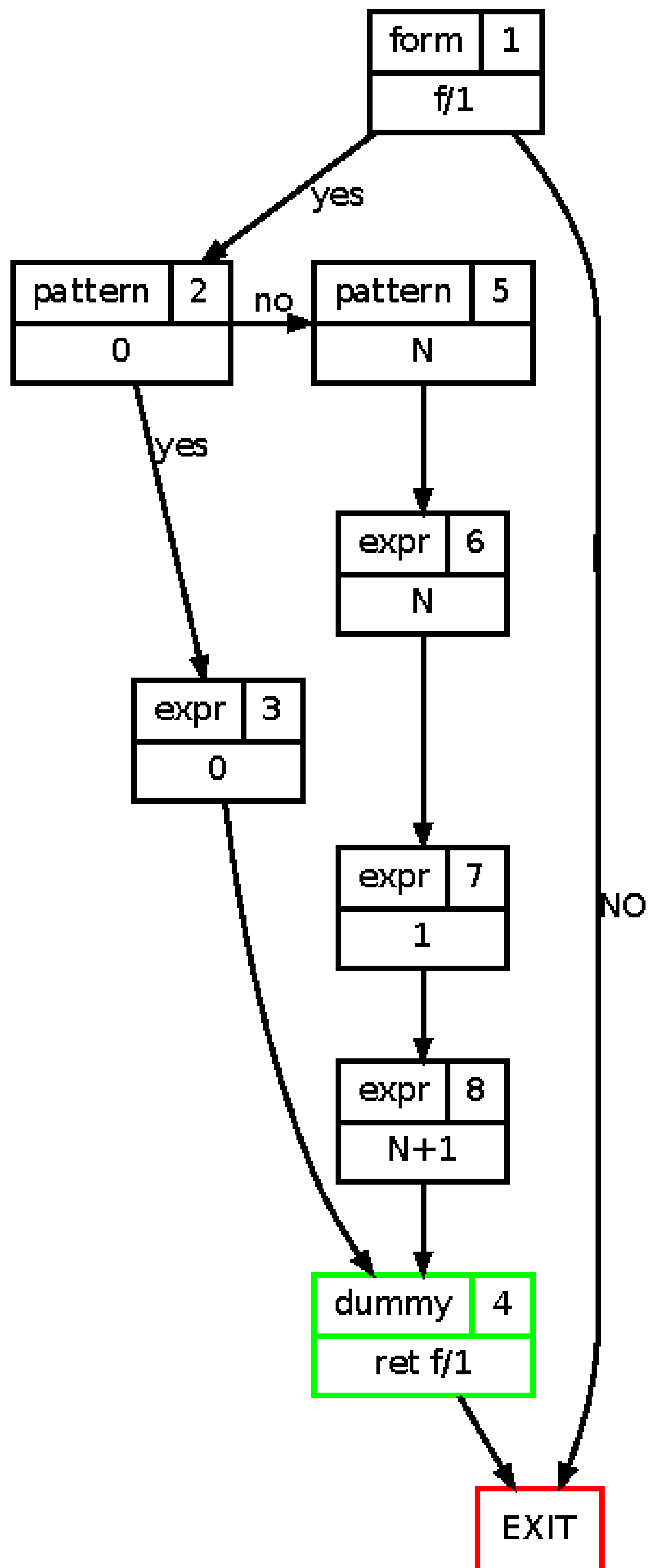
6 : { 1,2,3,4,5,6,7,8,EXIT }

7 : { 1,2,3,4,5,6,7,8,EXIT }

8 : { 1,2,3,4,5,6,7,8,EXIT }

EXIT : { EXIT }

Created by XMLmind XSL-FO Converter.


form
1
f/1
yes
pattern
2
0
no
pattern
5
N
yes
expr
6
N
expr
3
0
expr
7
1
NO
expr
8
N+1
dummy
4
ret f/1
EXIT

1.14. Example: Postdominators

Postdominator sets:

1 : { 1,EXIT }

2 : { 1,2,3,4,5,6,7,8,EXIT }

3 : { 1,2,3,4,5,6,7,8,EXIT }

4 : { 4,EXIT }

5 : { 1,2,3,4,5,6,7,8,EXIT }

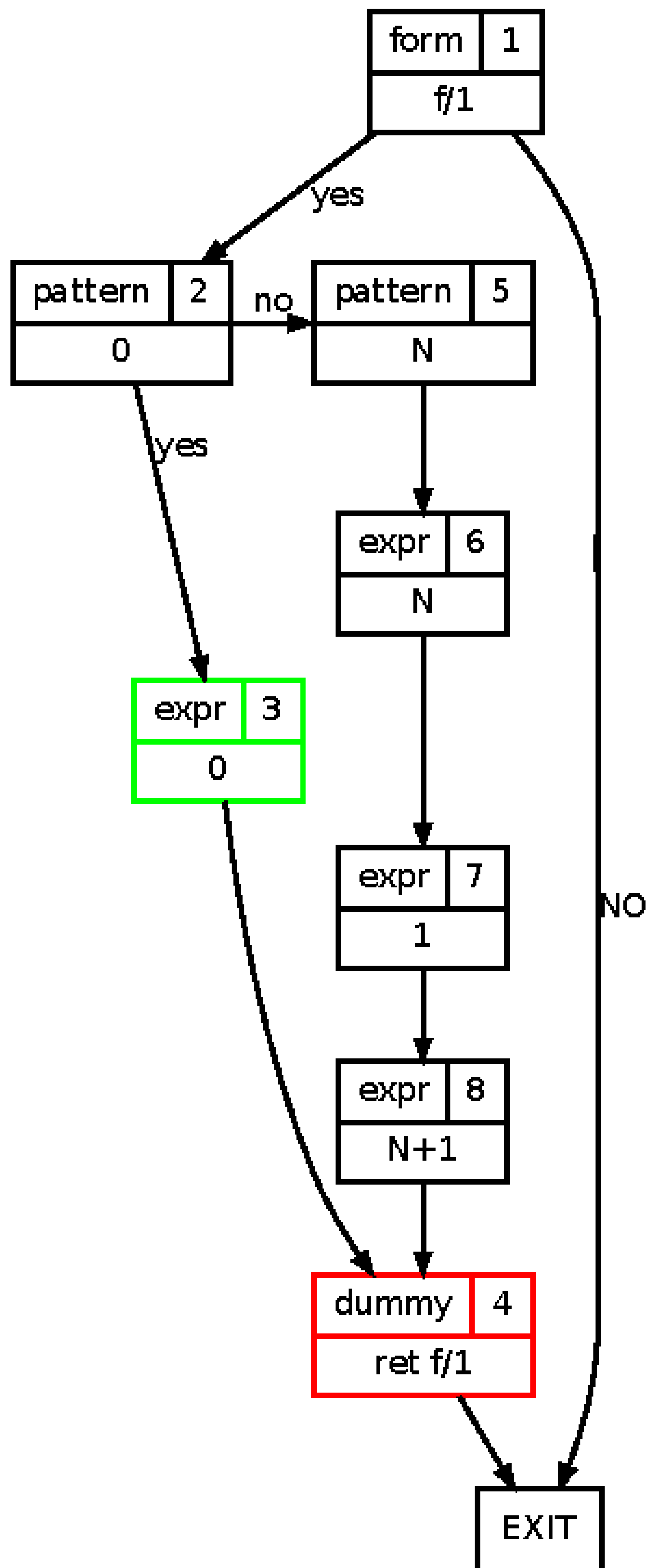
6 : { 1,2,3,4,5,6,7,8,EXIT }

7 : { 1,2,3,4,5,6,7,8,EXIT }

8 : { 1,2,3,4,5,6,7,8,EXIT }

EXIT : { EXIT }

Created by XMLmind XSL-FO Converter.


form
1
f/1
yes
pattern
2
0
no
pattern
5
N
yes
expr
6
N
expr
3
0
expr
7
1
NO
expr
8
N+1
dummy
4
ret f/1
EXIT

1.15. Example: Postdominators

Postdominator sets:

1 : { 1,EXIT }

2 : { 1,2,3,4,5,6,7,8,EXIT }

3 : { 4,EXIT } $\cup \{3\}$

4 : { 4,EXIT }

5 : { 1,2,3,4,5,6,7,8,EXIT }

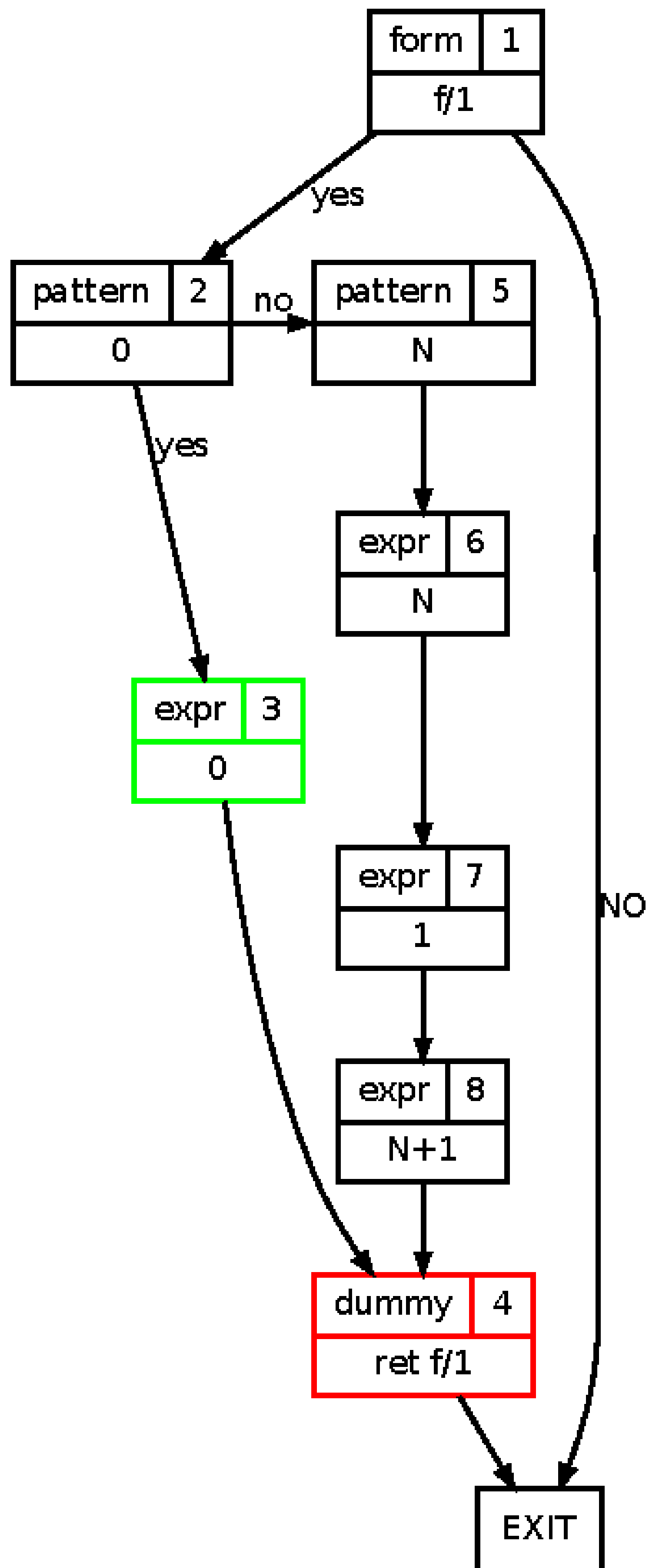
6 : { 1,2,3,4,5,6,7,8,EXIT }

7 : { 1,2,3,4,5,6,7,8,EXIT }

8 : { 1,2,3,4,5,6,7,8,EXIT }

EXIT : { EXIT }

Created by XMLmind XSL-FO Converter.


form
1
f/1
yes
pattern
2
0
no
pattern
5
N
yes
expr
6
N
expr
3
0
expr
7
1
NO
expr
8
N+1
dummy
4
ret f/1
EXIT

1.16. Example: Postdominators

Postdominator sets:

1 : { 1,EXIT }

2 : { 1,2,3,4,5,6,7,8,EXIT }

3 : { 3,4,EXIT }

4 : { 4,EXIT }

5 : { 1,2,3,4,5,6,7,8,EXIT }

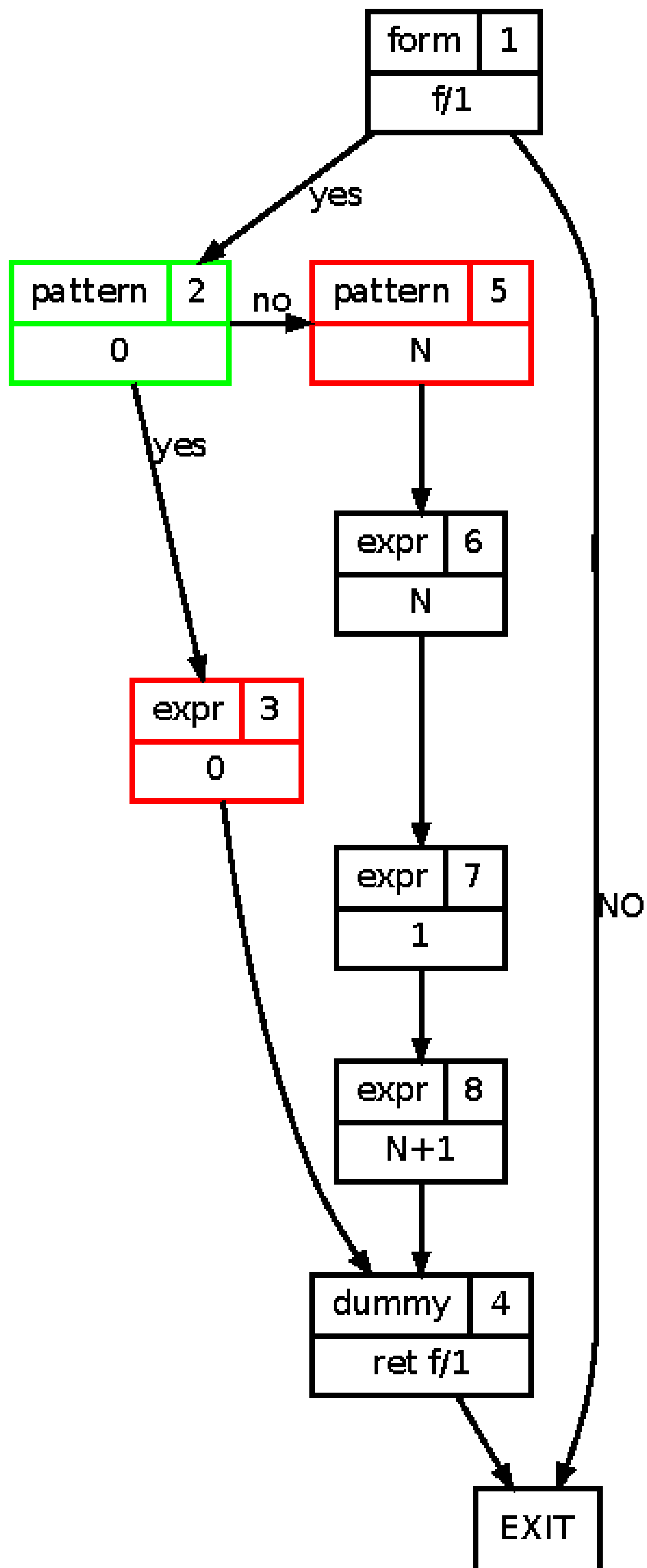
6 : { 1,2,3,4,5,6,7,8,EXIT }

7 : { 1,2,3,4,5,6,7,8,EXIT }

8 : { 1,2,3,4,5,6,7,8,EXIT }

EXIT : { EXIT }

Created by XMLmind XSL-FO Converter.


form
1
f/1
yes
pattern
2
0
no
pattern
5
N
yes
expr
6
N
expr
3
0
expr
7
1
NO
expr
8
N+1
dummy
4
ret f/1
EXIT

1.17. Example: Postdominators

Postdominator sets:

1 : { 1,EXIT }

2 : { 3^1,4,EXIT } $\cup \{2\}$

3 : { 3,4,EXIT }

4 : { 4,EXIT }

5 : { 1,2,3,4,5,6,7,8,EXIT }

6 : { 1,2,3,4,5,6,7,8,EXIT }

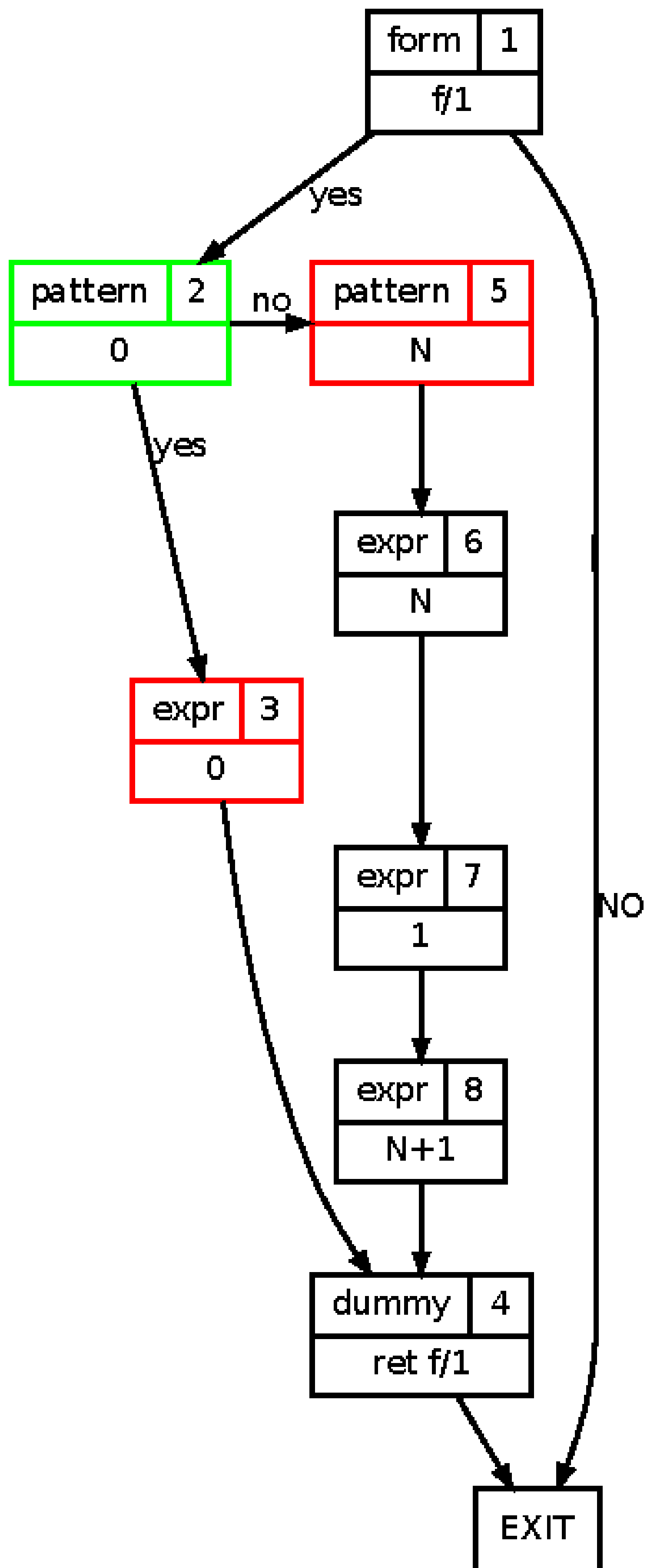
7 : { 1,2,3,4,5,6,7,8,EXIT }

8 : { 1,2,3,4,5,6,7,8,EXIT }

EXIT : { EXIT }

1 The node will be eliminated in a later step!

Created by XMLmind XSL-FO Converter.


form
1
f/1
yes
pattern
2
0
no
pattern
5
N
yes
expr
6
N
expr
3
0
expr
7
1
NO
expr
8
N+1
dummy
4
ret f/1
EXIT

1.18. Example: Postdominators

Postdominator sets:

1 : { 1,EXIT }

2 : { 2,3,4,EXIT }

3 : { 3,4,EXIT }

4 : { 4,EXIT }

5 : { 1,2,3,4,5,6,7,8,EXIT }

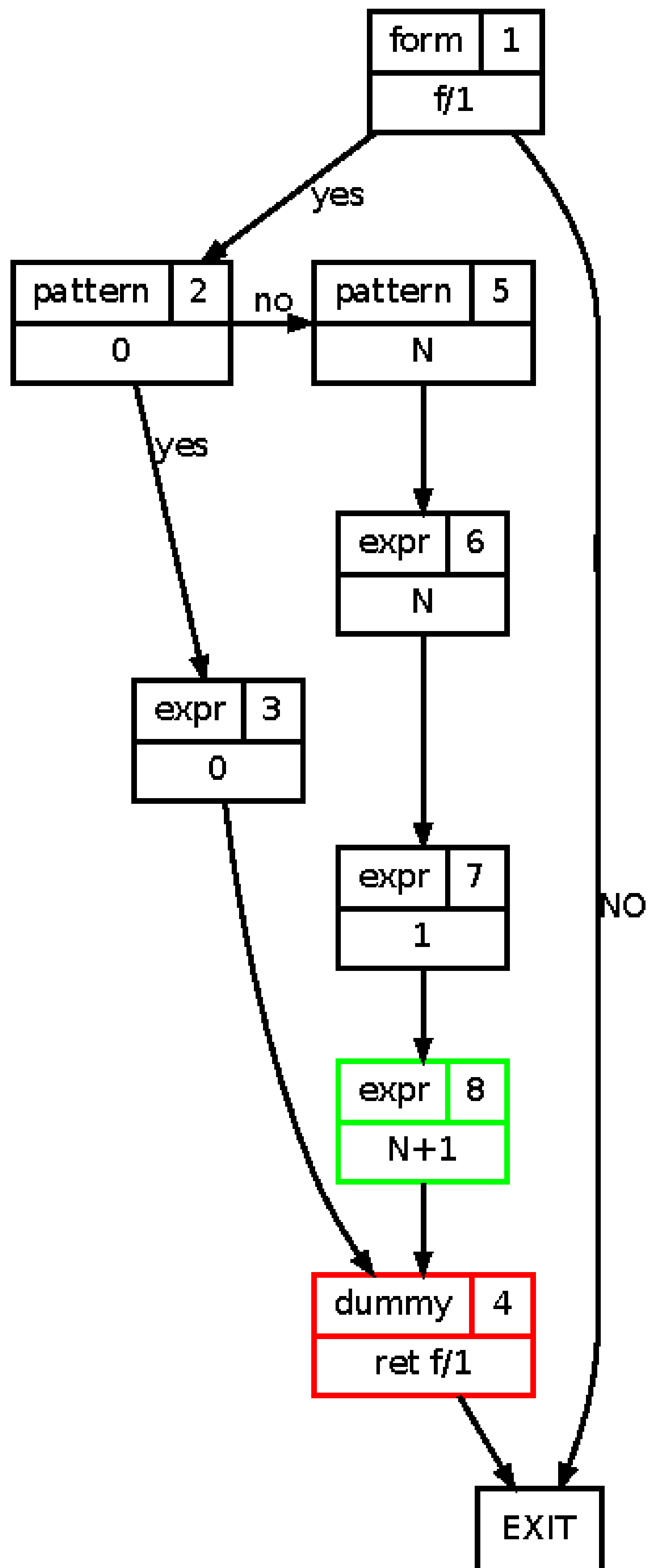
6 : { 1,2,3,4,5,6,7,8,EXIT }

7 : { 1,2,3,4,5,6,7,8,EXIT }

8 : { 1,2,3,4,5,6,7,8,EXIT }

EXIT : { EXIT }

Created by XMLmind XSL-FO Converter.


form
1
f/1
yes
pattern
2
0
no
pattern
5
N
yes
expr
6
N
expr
3
0
expr
7
1
NO
expr
8
N+1
dummy
4
ret f/1
EXIT

1.19. Example: Postdominators

Postdominator sets:

1 : { 1,EXIT }

2 : { 2,3,4,EXIT }

3 : { 3,4,EXIT }

4 : { 4,EXIT }

5 : { 1,2,3,4,5,6,7,8,EXIT }

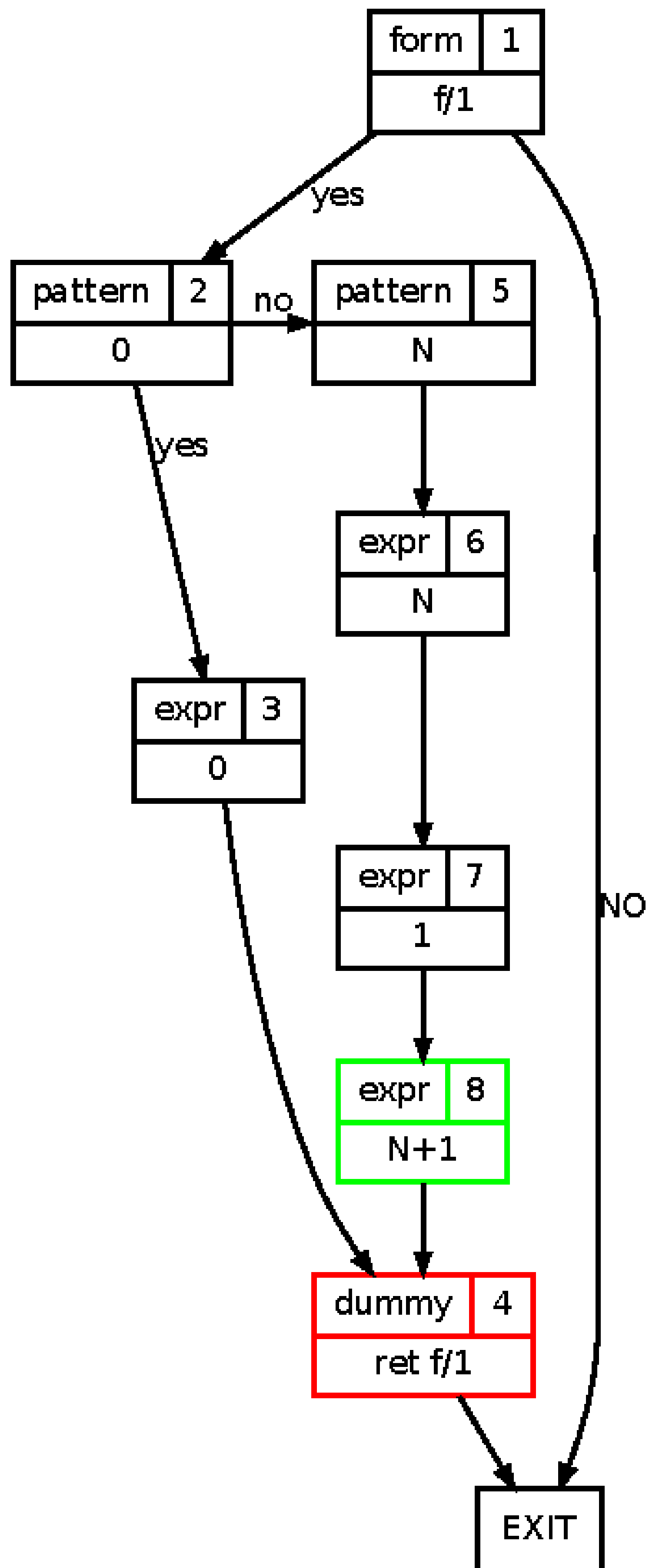
6 : { 1,2,3,4,5,6,7,8,EXIT }

7 : { 1,2,3,4,5,6,7,8,EXIT }

8 : { 4,EXIT } $\cup \{8\}$

EXIT : { EXIT }

Created by XMLmind XSL-FO Converter.


form
1
f/1
yes
NO
pattern
2
0
no
pattern
5
N
yes
expr
6
N
expr
3
0
expr
7
1
expr
8
N+1
dummy
4
ret f/1
EXIT

1.20. Example: Postdominators

Postdominator sets:

1 : { 1,EXIT }

2 : { 2,3,4,EXIT }

3 : { 3,4,EXIT }

4 : { 4,EXIT }

5 : { 1,2,3,4,5,6,7,8,EXIT }

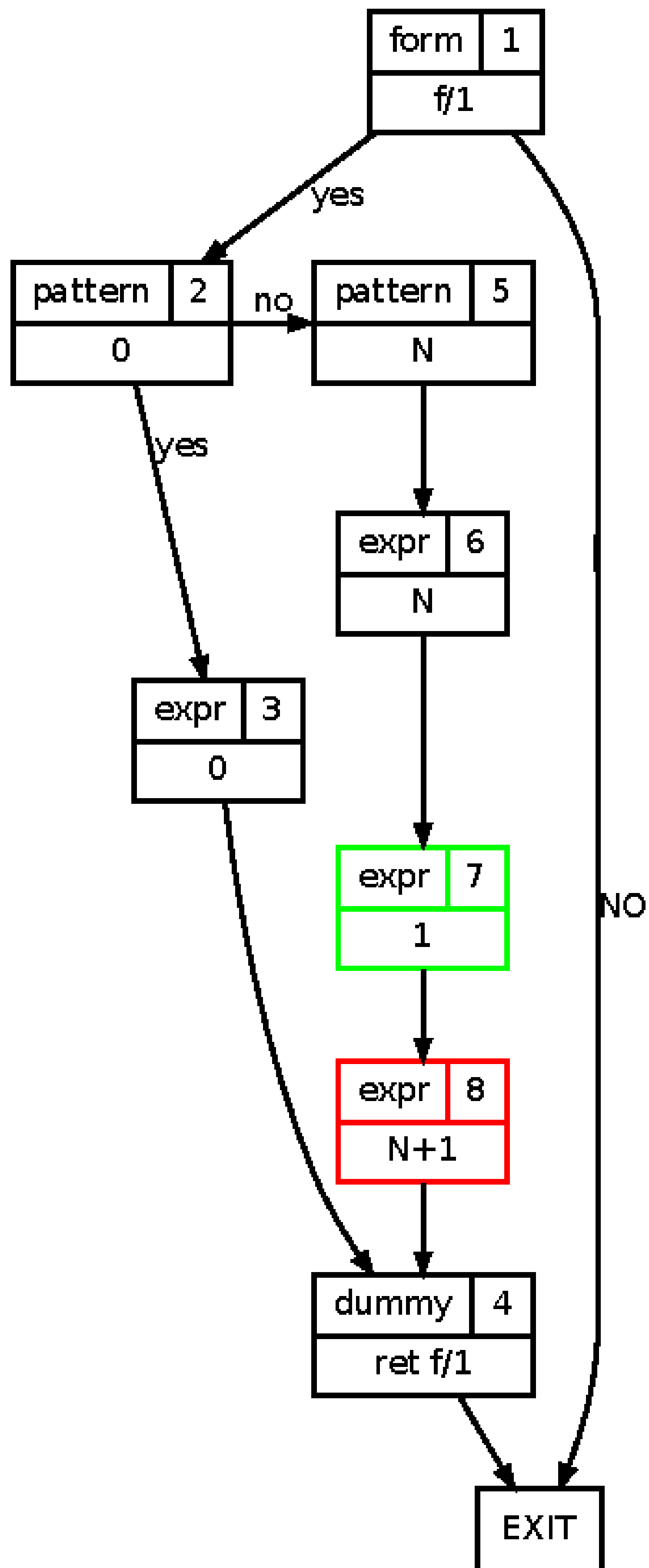
6 : { 1,2,3,4,5,6,7,8,EXIT }

7 : { 1,2,3,4,5,6,7,8,EXIT }

8 : { 4,8,EXIT }

EXIT : { EXIT }

Created by XMLmind XSL-FO Converter.


form
1
f/1
yes
NO
pattern
2
0
no
pattern
5
N
yes
expr
6
N
expr
3
0
expr
7
1
expr
8
N+1
dummy
4
ret f/1
EXIT

1.21. Example: Postdominators

Postdominator sets:

1 : { 1,EXIT }

2 : { 2,3,4,EXIT }

3 : { 3,4,EXIT }

4 : { 4,EXIT }

5 : { 1,2,3,4,5,6,7,8,EXIT }

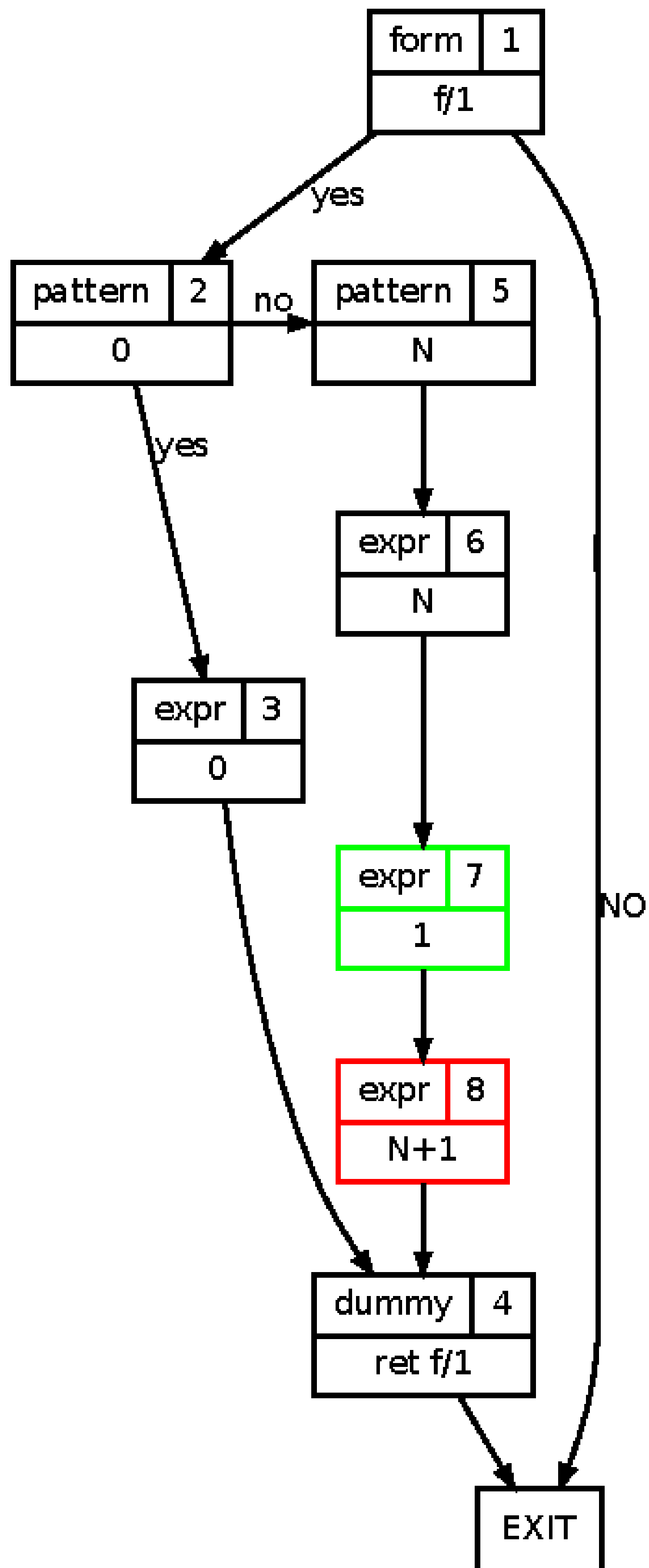
6 : { 1,2,3,4,5,6,7,8,EXIT }

7 : { 4,8,EXIT } $\cup \{7\}$

8 : { 4,8,EXIT }

EXIT : { EXIT }

Created by XMLmind XSL-FO Converter.


form
1
f/1
yes
NO
pattern
2
0
no
pattern
5
N
yes
expr
6
N
expr
3
0
expr
7
1
expr
8
N+1
dummy
4
ret f/1
EXIT

1.22. Example: Postdominators

Postdominator sets:

1 : { 1,EXIT }

2 : { 2,3,4,EXIT }

3 : { 3,4,EXIT }

4 : { 4,EXIT }

5 : { 1,2,3,4,5,6,7,8,EXIT }

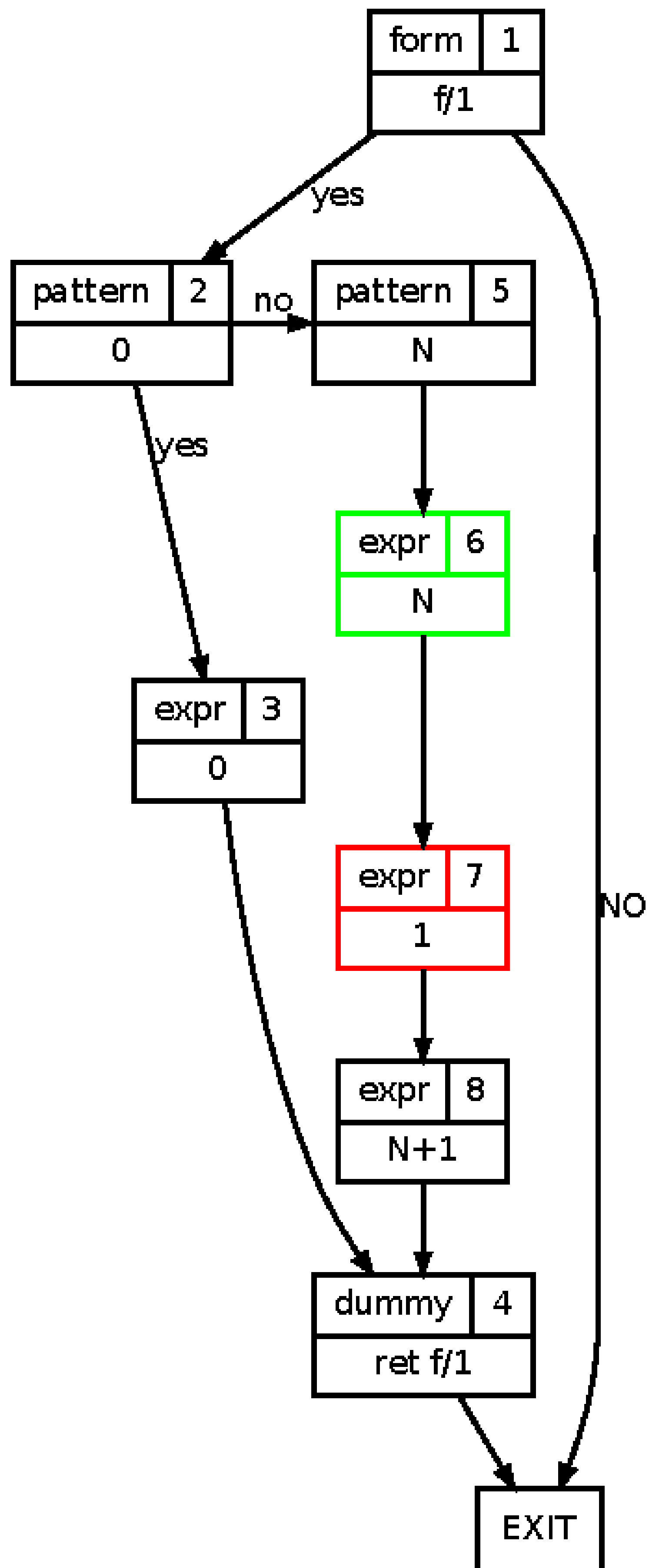
6 : { 1,2,3,4,5,6,7,8,EXIT }

7 : { 4,7,8,EXIT }

8 : { 4,8,EXIT }

EXIT : { EXIT }

Created by XMLmind XSL-FO Converter.


form
1
f/1
yes
pattern
2
0
no
pattern
5
N
yes
expr
6
N
expr
3
0
expr
7
1
NO
expr
8
N+1
dummy
4
ret f/1
EXIT

1.23. Example: Postdominators

Postdominator sets:

1 : { 1,EXIT }

2 : { 2,3,4,EXIT }

3 : { 3,4,EXIT }

4 : { 4,EXIT }

5 : { 1,2,3,4,5,6,7,8,EXIT }

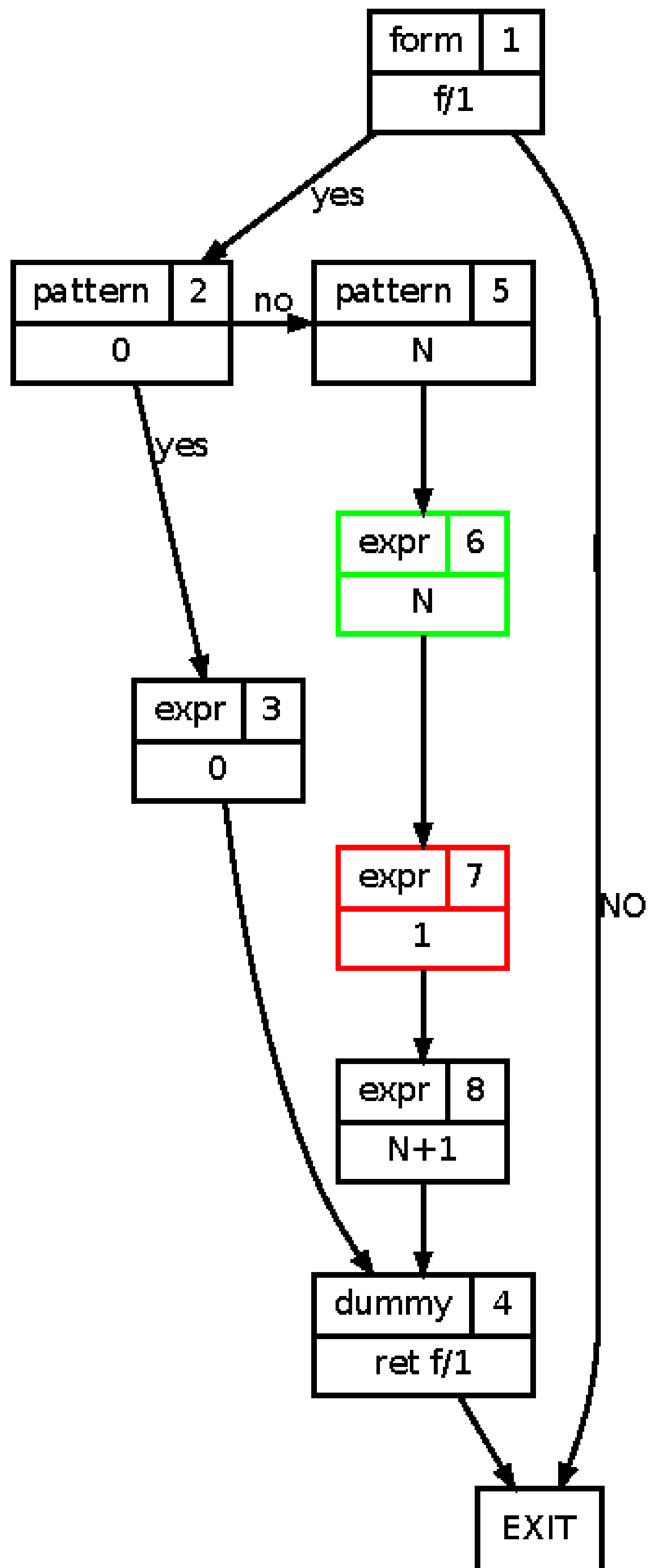
6 : { 4,7,8,EXIT } $\cup \{6\}$

7 : { 4,7,8,EXIT }

8 : { 4,8,EXIT }

EXIT : { EXIT }

Created by XMLmind XSL-FO Converter.


form
1
f/1
yes
pattern
2
0
no
pattern
5
N
yes
expr
6
N
expr
3
0
expr
7
1
NO
expr
8
N+1
dummy
4
ret f/1
EXIT

1.24. Example: Postdominators

Postdominator sets:

1 : { 1,EXIT }

2 : { 2,3,4,EXIT }

3 : { 3,4,EXIT }

4 : { 4,EXIT }

5 : { 1,2,3,4,5,6,7,8,EXIT }

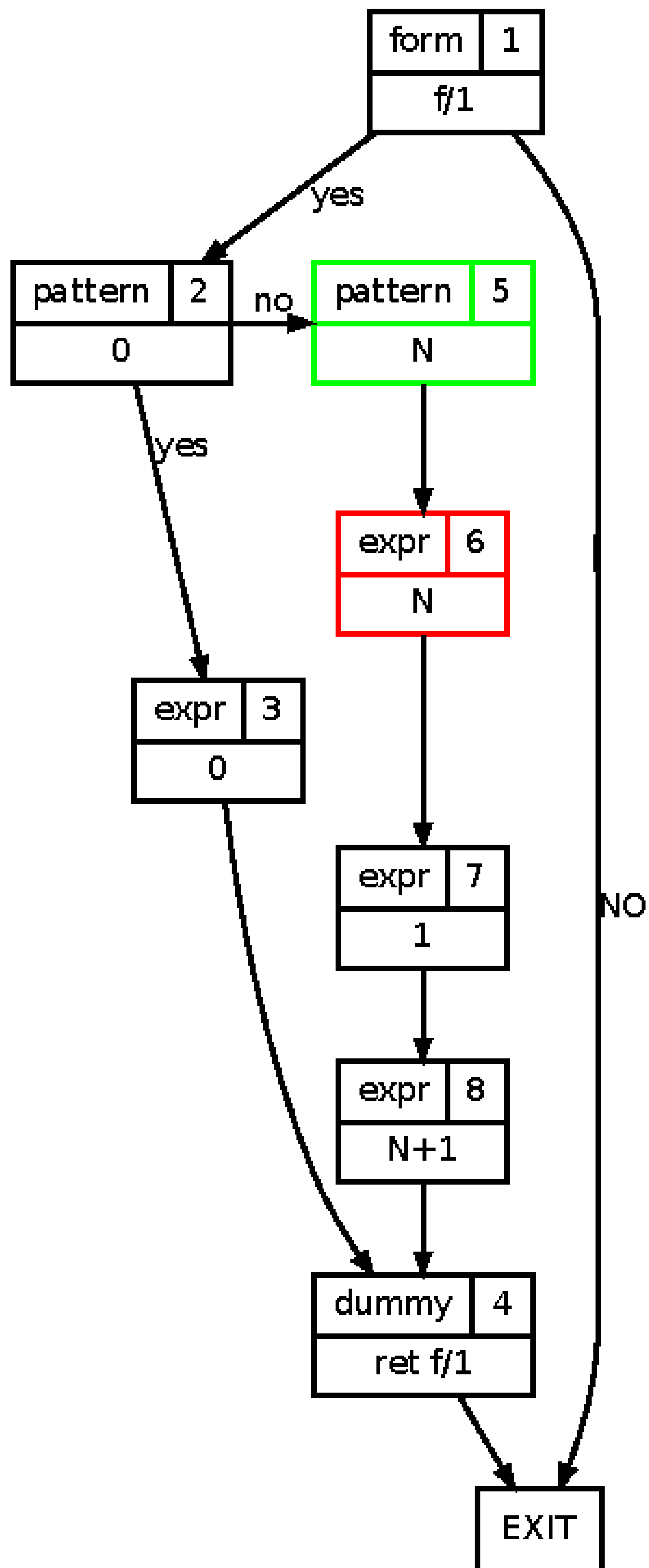
6 : { 4,6,7,8,EXIT }

7 : { 4,7,8,EXIT }

8 : { 4,8,EXIT }

EXIT : { EXIT }

Created by XMLmind XSL-FO Converter.


form
1
f/1
yes
pattern
2
0
no
pattern
5
N
yes
expr
6
N
expr
3
0
expr
7
1
NO
expr
8
N+1
dummy
4
ret f/1
EXIT

1.25. Example: Postdominators

Postdominator sets:

1 : { 1,EXIT }

2 : { 2,3,4,EXIT }

3 : { 3,4,EXIT }

4 : { 4,EXIT }

5 : { 4,6,7,8,EXIT } $\cup \{5\}$

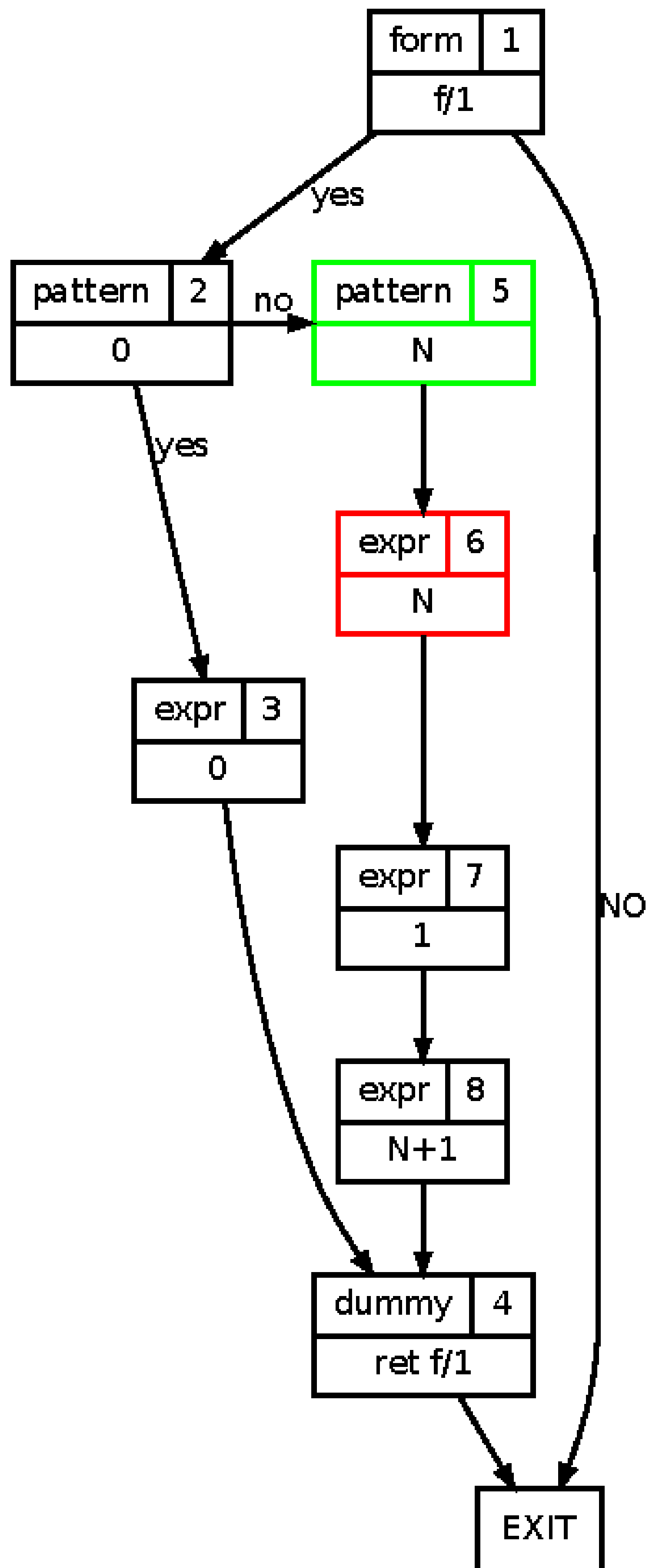
6 : { 4,6,7,8,EXIT }

7 : { 4,7,8,EXIT }

8 : { 4,8,EXIT }

EXIT : { EXIT }

Created by XMLmind XSL-FO Converter.


form
1
f/1
yes
NO
pattern
2
0
no
pattern
5
N
yes
expr
6
N
expr
3
0
expr
7
1
expr
8
N+1
dummy
4
ret f/1
EXIT

1.26. Example: Postdominators

Postdominator sets:

1 : { 1,EXIT }

2 : { 2,3,4,EXIT }

3 : { 3,4,EXIT }

4 : { 4,EXIT }

5 : { 4,5,6,7,8,EXIT }

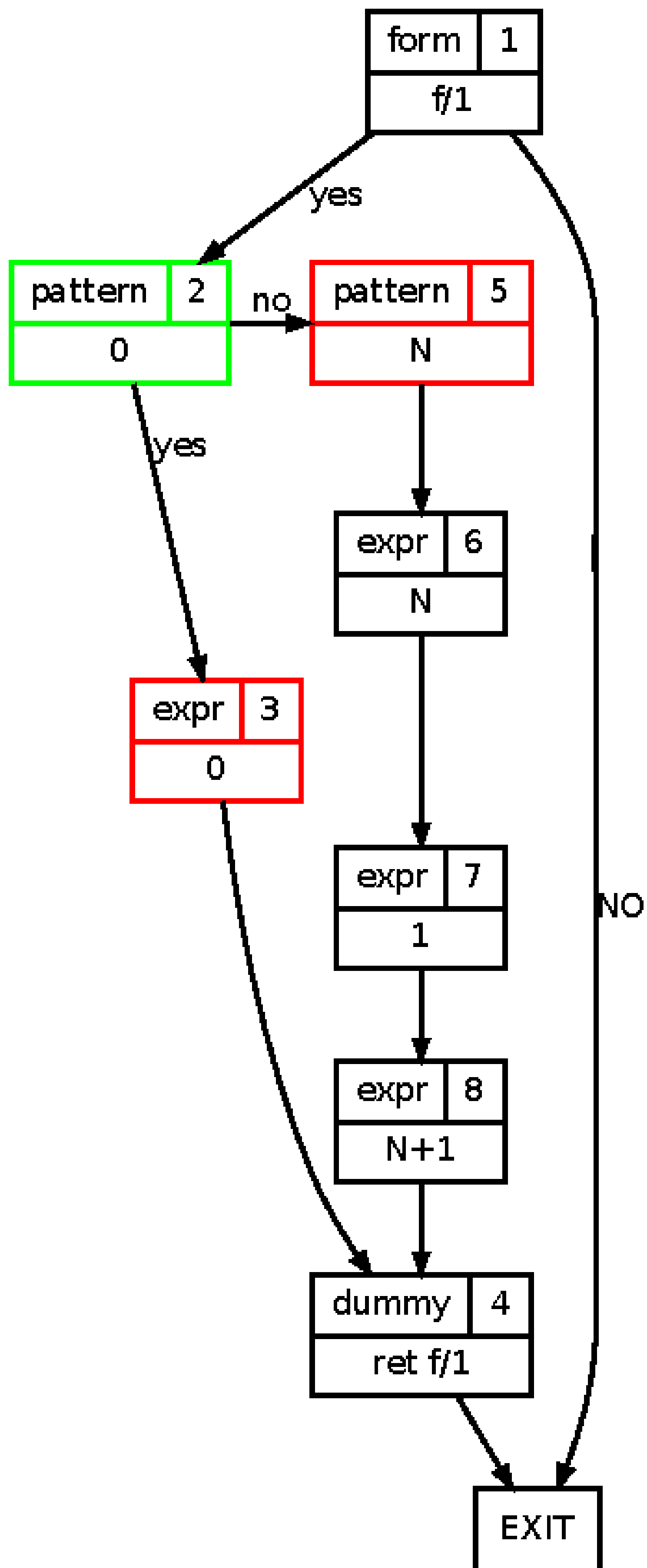
6 : { 4,6,7,8,EXIT }

7 : { 4,7,8,EXIT }

8 : { 4,8,EXIT }

EXIT : { EXIT }

Created by XMLmind XSL-FO Converter.


form
1
f/1
yes
pattern
2
0
no
pattern
5
N
yes
expr
6
N
expr
3
0
expr
7
1
NO
expr
8
N+1
dummy
4
ret f/1
EXIT

1.27. Example: Postdominators

Final postdominator sets:

1 : { 1, EXIT }

2 : { 2, 4, EXIT }

3 : { 3, 4, EXIT }

4 : { 4, EXIT }

5 : { 4, 5, 6, 7, 8, EXIT }

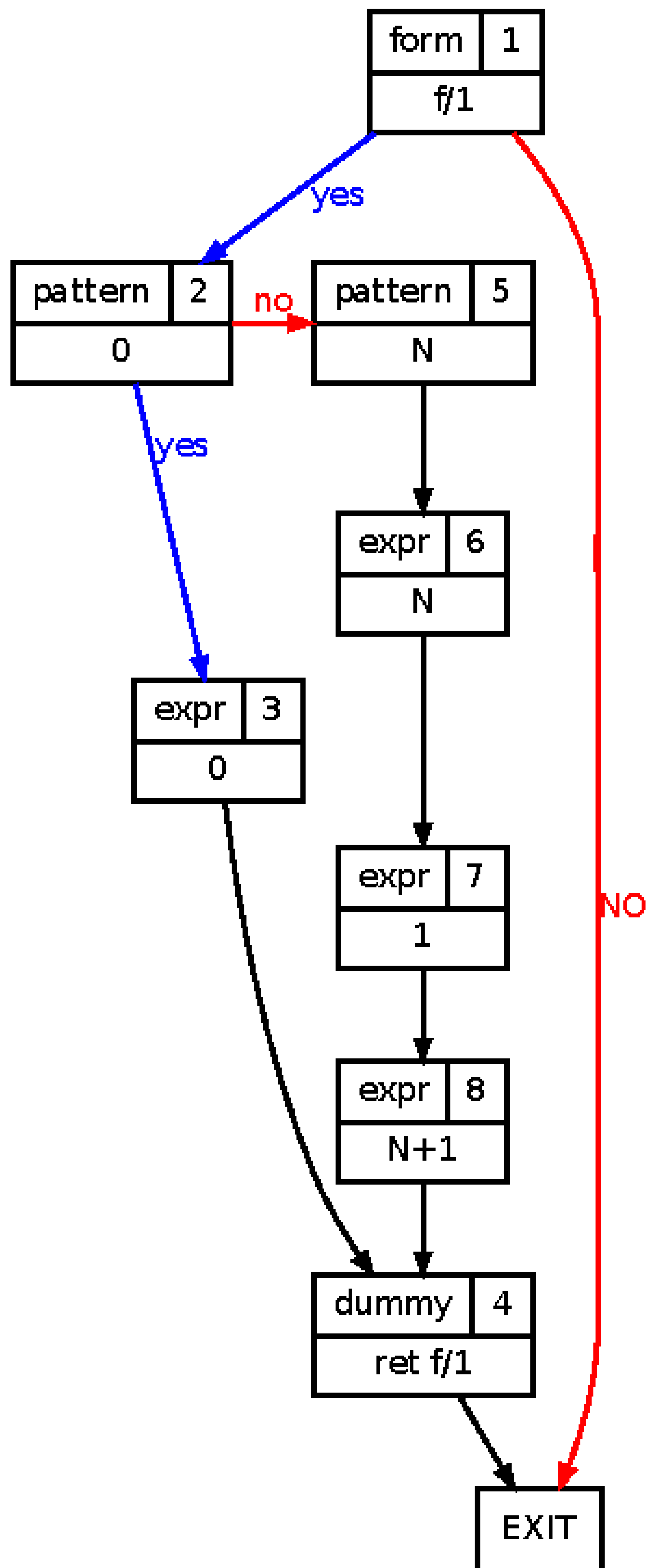
6 : { 4, 6, 7, 8, EXIT }

7 : { 4, 7, 8, EXIT }

8 : { 4, 8, EXIT }

EXIT : { EXIT }

Created by XMLmind XSL-FO Converter.


form
1
f/1
yes
NO
pattern
2
0
no
pattern
5
N
yes
expr
6
N
expr
3
0
expr
7
1
expr
8
N+1
dummy
4
ret f/1
EXIT

1.28. Example: Remarks on Postdominator Calculation

- These steps show one of the possible evaluation orders of the given algorithm.
- It can be seen that the most optimal evaluation of the algorithm is if we start from the exit node and systematically proceed to its ancestors.

1.29. Example: Immediate Postdominators

Initialised immediate postdominator sets:

1, EXIT }	→	**1** : { EXIT }
2, 4, EXIT }	→	**2** : { 4, EXIT }
3, 4, EXIT }	→	**3** : { 4, EXIT }
4, EXIT }	→	**4** : { EXIT }
4, 5, 6, 7, 8, EXIT }	→	**5** : { 4, 6, 7, 8, E
4, 6, 7, 8, EXIT }	→	**6** : { 4, 7, 8, EXIT
4, 7, 8, EXIT }	→	**7** : { 4, 8, EXIT }
4, 8, EXIT }	→	**8** : { 4, EXIT }
: { EXIT }	→	**EXIT** : ∅

1.30. Example: Immediate Postdominators

Immediate postdominator sets:

1 : { EXIT }

2 : { 4, EXIT }

3 : { 4, EXIT }

4 : { EXIT }

5 : { 4, 6, 7, 8, EXIT }

6 : { 4, 7, 8, EXIT }

7 : { 4, 8, EXIT }

8 : { 4, EXIT }

EXIT : $\emptyset$

EXIT pdom 4

4 pdom EXIT

To delete: { EXIT }

1.31. Example: Immediate Postdominators

Immediate postdominator sets:

Created by XMLmind XSL-FO Converter.

1 : { EXIT }

2 : { 4, EXIT }

3 : { 4, EXIT }

4 : { EXIT }

5 : { 4, 6, 7, 8, EXIT }

6 : { 4, 7, 8, EXIT }

7 : { 4, 8, EXIT }

8 : { 4, EXIT }

EXIT : $\emptyset$

EXIT pdom 4

4 pdom EXIT

To delete: { EXIT }

1.32. Example: Immediate Postdominators

Immediate postdominator sets:

1 : { EXIT }

2 : { 4, EXIT }

3 : { 4, EXIT }

4 : { EXIT }

5 : { 4, 6, 7, 8, EXIT }

6 : { 4, 7, 8, EXIT }

7 : { 4, 8, EXIT }

8 : { 4, EXIT }

EXIT : $\emptyset$

EXIT pdom 4

4 pdom EXIT

To delete: { EXIT }

1.33. Example: Immediate Postdominators

Immediate postdominator sets:

1 : { EXIT }

2 : { 4, EXIT }

3 : { 4, EXIT }

4 : { EXIT }

Created by XMLmind XSL-FO Converter.

5 : { 4, 6, 7, 8, EXIT }

6 : { 4, 7, 8, EXIT }

7 : { 4, 8, EXIT }

8 : { 4, EXIT }

EXIT : $\emptyset$

EXIT pdom 4

4 pdom EXIT

To delete: { EXIT }

1.34. Example: Immediate Postdominators

Immediate postdominator sets:

1 : { EXIT }

2 : { 4, EXIT }

3 : { 4, EXIT }

4 : { EXIT }

5 : { 4, 6, 7, 8, EXIT }

6 : { 4, 7, 8, EXIT }

7 : { 4, 8, EXIT }

8 : { 4, EXIT }

EXIT : $\emptyset$

4 pdom 6 – **6 pdom 4**

4 pdom 7 – **7 pdom 4**

4 pdom 8 – **8 pdom 4**

4 pdom EXIT – **EXIT pdom 4**

6 pdom 7 – **7 pdom 6**

6 pdom 8 – **8 pdom 6**

6 pdom EXIT – **EXIT pdom 6**

7 pdom 8 – **8 pdom 7**

7 pdom EXIT – **EXIT pdom 7**

8 pdom EXIT – **EXIT pdom 8**

To delete: { 4, 7, 8, EXIT }

1.35. Example: Immediate Postdominators

Immediate postdominator sets:

Created by XMLmind XSL-FO Converter.

1 : { EXIT }

2 : { 4, EXIT }

3 : { 4, EXIT }

4 : { EXIT }

5 : { 4, 6, 7, 8, EXIT }

6 : { 4, 7, 8, EXIT }

7 : { 4, 8, EXIT }

8 : { 4, EXIT }

EXIT : $\emptyset$

4 pdom 6 – 6 pdom 4

4 pdom 7 – 7 pdom 4

4 pdom 8 – 8 pdom 4

4 pdom EXIT – EXIT pdom 4

6 pdom 7 – 7 pdom 6

6 pdom 8 – 8 pdom 6

6 pdom EXIT – EXIT pdom 6

7 pdom 8 – 8 pdom 7

7 pdom EXIT – EXIT pdom 7

8 pdom EXIT – EXIT pdom 8

To delete: { 4, 7, 8, EXIT }

1.36. Example: Immediate Postdominators

Immediate postdominator sets:

1 : { EXIT }

2 : { 4, EXIT }

3 : { 4, EXIT }

4 : { EXIT }

5 : { 4, 6, 7, 8, EXIT }

6 : { 4, 7, 8, EXIT }

7 : { 4, 8, EXIT }

8 : { 4, EXIT }

EXIT : $\emptyset$

4 pdom 7 – 7 pdom 4

Created by XMLmind XSL-FO Converter.

4 pdom 8 – 8 pdom 4

4 pdom EXIT – EXIT pdom 4

7 pdom 8 – 8 pdom 7

7 pdom EXIT – EXIT pdom 7

8 pdom EXIT – EXIT pdom 8

To delete: { 4, 8, EXIT }

1.37. Example: Immediate Postdominators

Immediate postdominator sets:

1 : { EXIT }

2 : { 4, EXIT }

3 : { 4, EXIT }

4 : { EXIT }

5 : { 4, 6, 7, 8, EXIT }

6 : { 4, 7, 8, EXIT }

7 : { 4, 8, EXIT }

8 : { 4, EXIT }

EXIT : $\emptyset$

4 pdom 7 – 7 pdom 4

4 pdom 8 – 8 pdom 4

4 pdom EXIT – EXIT pdom 4

7 pdom 8 – 8 pdom 7

7 pdom EXIT – EXIT pdom 7

8 pdom EXIT – EXIT pdom 8

To delete: { 4, 8, EXIT }

1.38. Example: Immediate Postdominators

Immediate postdominator sets:

1 : { EXIT }

2 : { 4, EXIT }

3 : { 4, EXIT }

4 : { EXIT }

5 : { 4, 6, 7, 8, EXIT }

6 : { 4, 7, 8, EXIT }

Created by XMLmind XSL-FO Converter.

7 : { 4, 8, EXIT }

8 : { 4, EXIT }

EXIT : ∅

4 pdom 8 – 8 pdom 4

4 pdom EXIT – EXIT pdom 4

8 pdom EXIT – EXIT pdom 8

To delete: { 4, EXIT }

1.39. Example: Immediate Postdominators

Immediate postdominator sets:

1 : { EXIT }

2 : { 4, EXIT }

3 : { 4, EXIT }

4 : { EXIT }

5 : { 4, 6, 7, 8, EXIT }

6 : { 4, 7, 8, EXIT }

7 : { 4, 8, EXIT }

8 : { 4, EXIT }

EXIT : ∅

4 pdom 8 – 8 pdom 4

4 pdom EXIT – EXIT pdom 4

8 pdom EXIT – EXIT pdom 8

To delete: { 4, EXIT }

1.40. Example: Immediate Postdominators

Immediate postdominator sets:

1 : { EXIT }

2 : { 4, EXIT }

3 : { 4, EXIT }

4 : { EXIT }

5 : { 4, 6, 7, 8, EXIT }

6 : { 4, 7, 8, EXIT }

7 : { 4, 8, EXIT }

8 : { 4, EXIT }

Created by XMLmind XSL-FO Converter.

EXIT : $\emptyset$

4 pdom EXIT – EXIT pdom 4

To delete: { EXIT }

1.41. Example: Immediate Postdominators

Immediate postdominator sets:

1 : { EXIT }

2 : { 4, EXIT }

3 : { 4, EXIT }

4 : { EXIT }

5 : { 4, 6, 7, 8, EXIT }

6 : { 4, 7, 8, EXIT }

7 : { 4, 8, EXIT }

8 : { 4, EXIT }

EXIT : $\emptyset$

4 pdom EXIT – EXIT pdom 4

To delete: { EXIT }

1.42. Example: Postdominator Tree

1 : { EXIT }

2 : { 4, EXIT }

3 : { 4, EXIT }

4 : { EXIT }

5 : { 4, 6, 7, 8, EXIT }

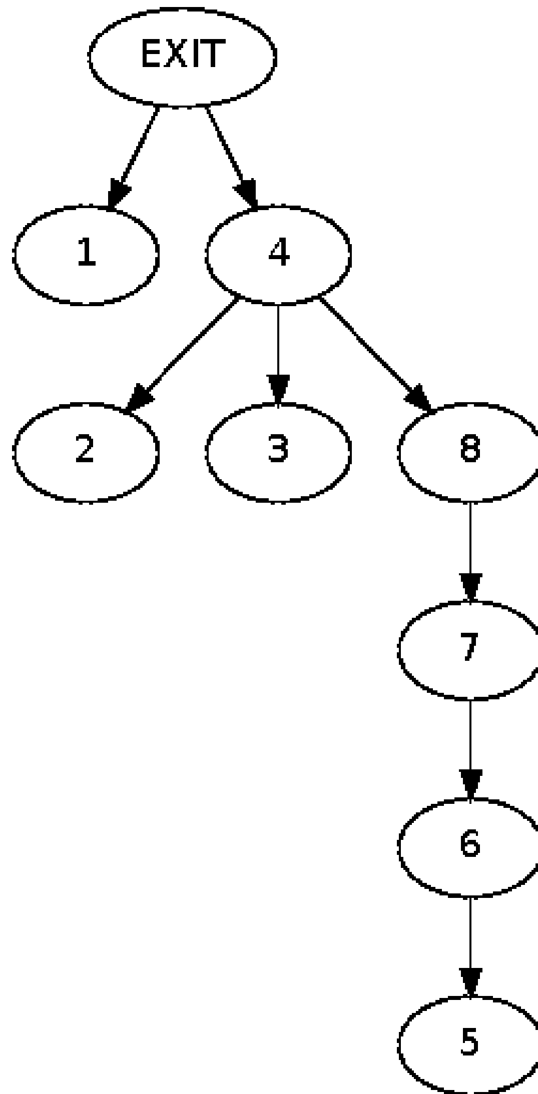
6 : { 4, 7, 8, EXIT }

7 : { 4, 8, EXIT }

8 : { 4, EXIT }

EXIT : $\emptyset$

Created by XMLmind XSL-FO Converter.



Created by XMLmind XSL-FO Converter.

Chapter 10. Lecture 10

1. Control-dependence

1.1. Control-Dependence Analysis

- CDG: direct control dependencies
- To determine control dependencies we need the CFG and the Postdominator Tree
- Select the edges ($a \rightarrow b$) from the CFG such that node a is not postdominated by node b
- Let denote lca the least common ancestor of a and b in the PDT
- Since the CFG is intrafunctional the CDG is defined only for functions separately

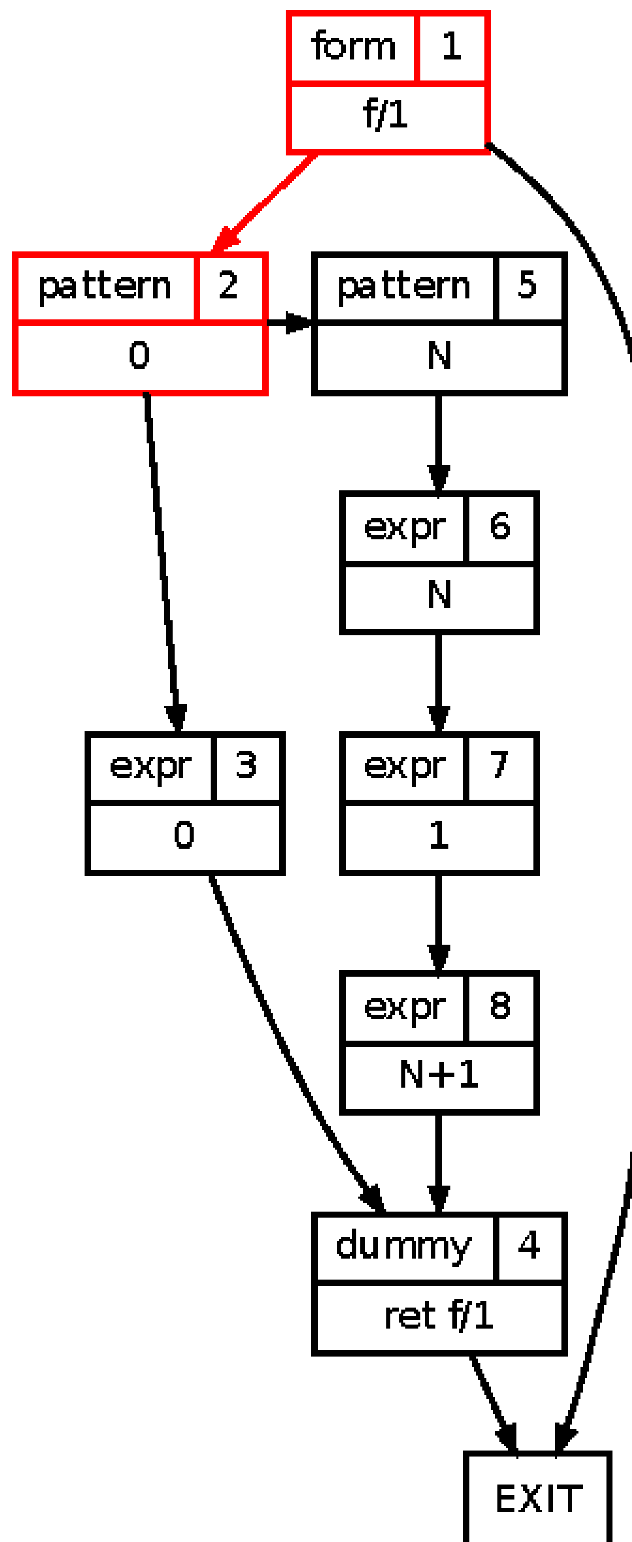
1.2. Control-Dependence Analysis

- The lca is the parent of the node a or it is a itself
  - If lca is parent of node a then every node on the path from lca to b , including b , but not lca , are control dependent on a
  - If $lca = a$ then every node on the path from a to b , including a and b , are control dependent on a (loop)
- Based on this we can determine the direct control dependencies

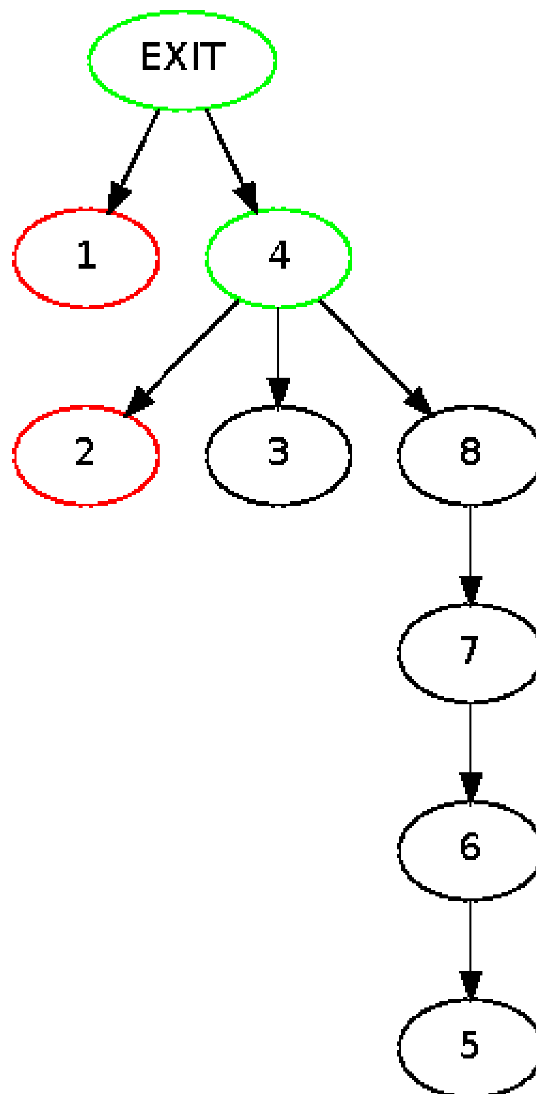
1.3. Example (1)

1: 2, 4

Created by XMLmind XSL-FO Converter.



Created by XMLmind XSL-FO Converter.

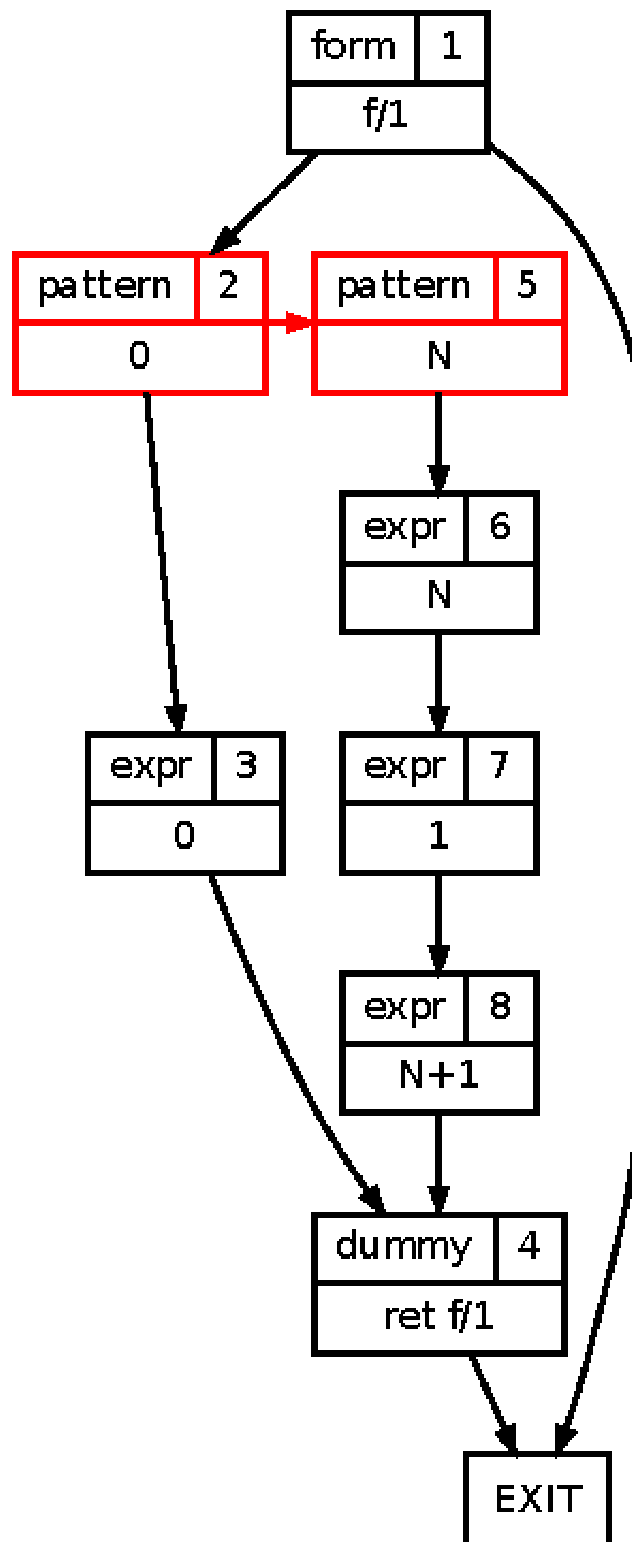


1.4. Example (2)

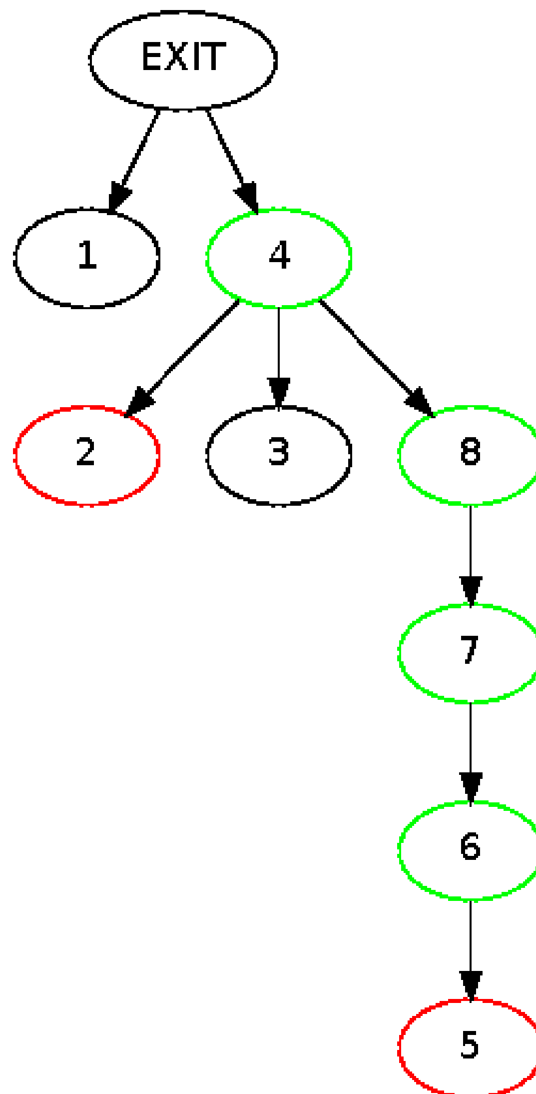
1: 2, 4

2: 5, 6, 7, 8

Created by XMLmind XSL-FO Converter.



Created by XMLmind XSL-FO Converter.

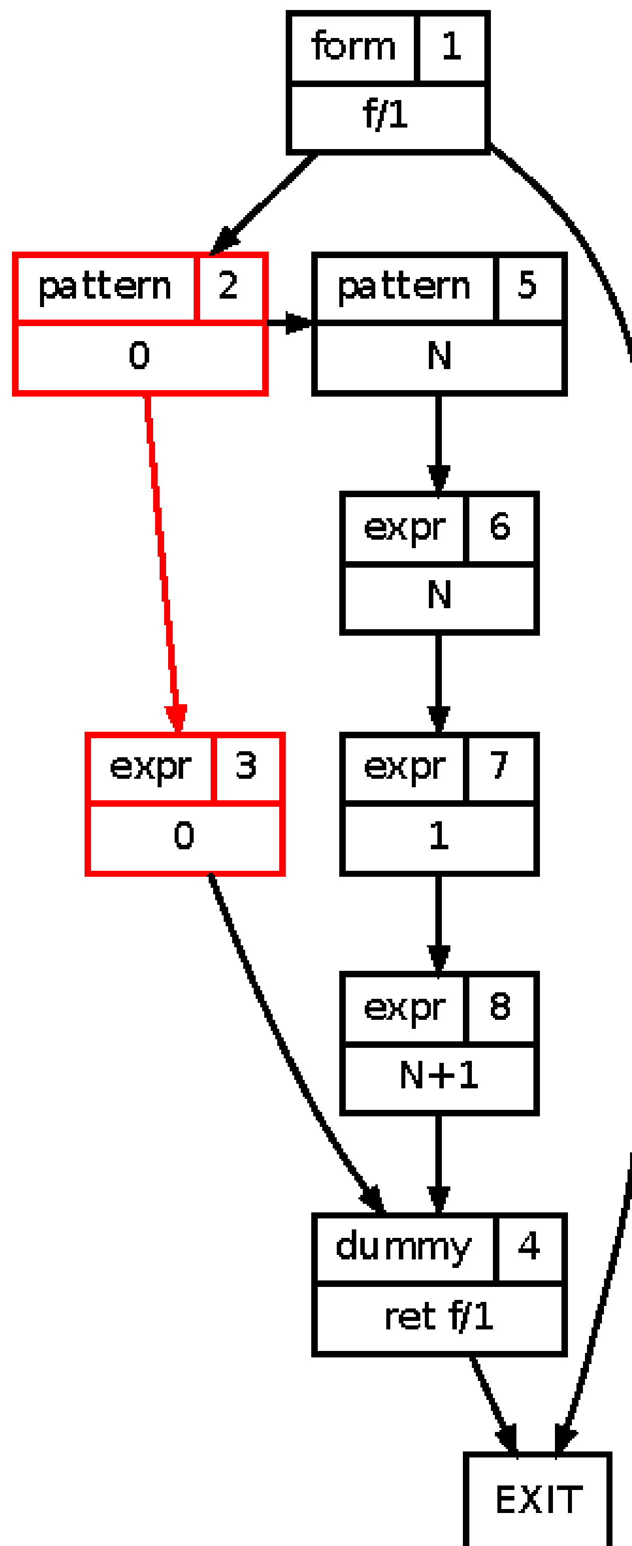


1.5. Example (3)

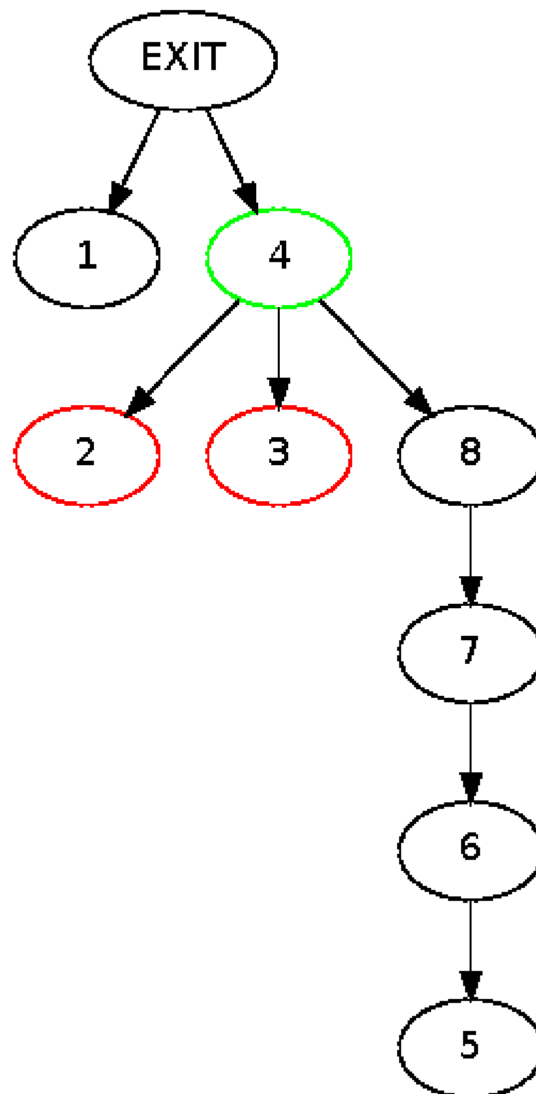
1: 2, 4

2: 3, 5, 6, 7, 8

Created by XMLmind XSL-FO Converter.



Created by XMLmind XSL-FO Converter.



1.6. Example (4)

1: 2, 4

2: 3, 5, 6, 7, 8

$1 \to 2$

$1 \to 4$

$2 \to 3$

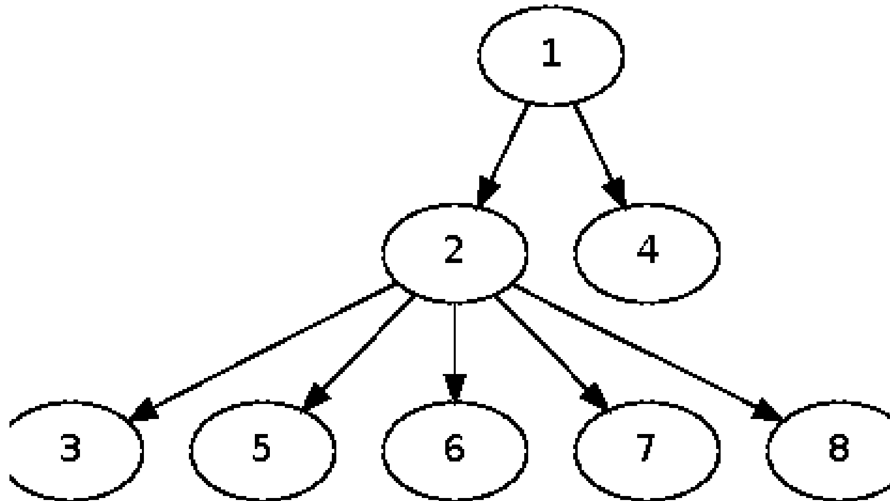
$2 \to 5$

$2 \to 6$

Created by XMLmind XSL-FO Converter.

$2 \rightarrow 7$

$2 \rightarrow 8$



1.7. Control Dependence Calculating Algorithm

- The previous analysis is defined for separate functions
- The presented calculation does not take into account the function calls, message passing and receiving
- We apply the above mentioned algorithm for every function involved in the analysis and compose the result where the previously mentioned dependencies are considered

1.8. Extending the Control Dependence Calculating Algorithm (1)

- Function calls, message receiving and sending expressions are collected
- Every function is examined whether it may fail or not
- Function may potentially fail if:
  - has no exhaustive patterns
  - contains an expression that may fail
  - throws an exception
- Failing function may affect other expression in the evaluation order, thus the expressions following the applications of potentially failing function are dependent on the application
- This information is used during the composing stage

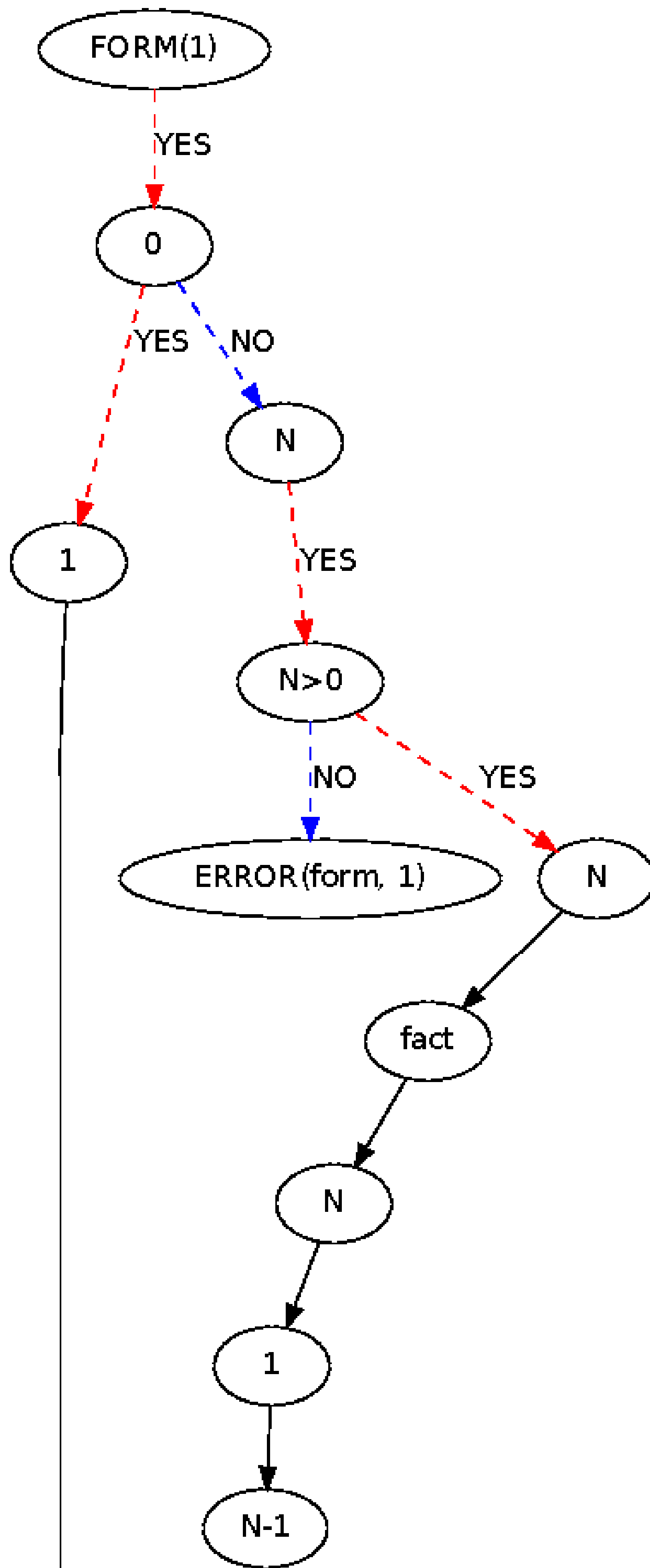
1.9. Example (CFG of Factorial Function)

[c]

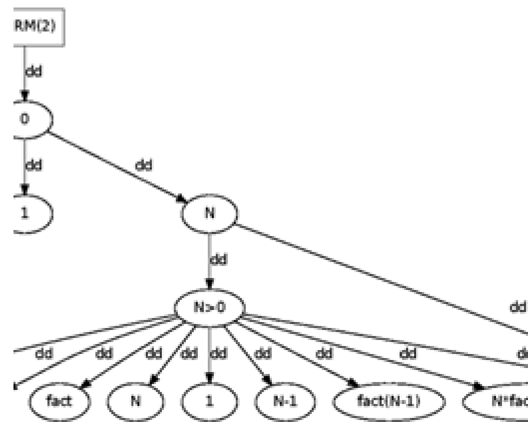
Created by XMLmind XSL-FO Converter.

```
  fact(0) ->
    1;
fact(N) when N > 0 ->
    N * fact(N-1).
```


Created by XMLmind XSL-FO Converter.

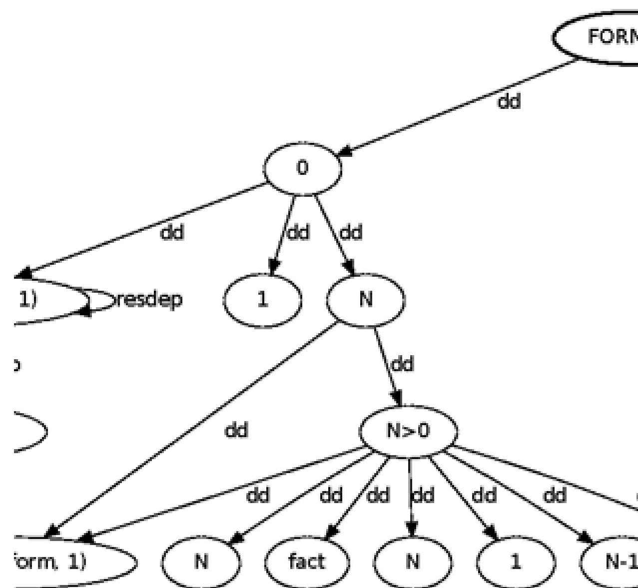

FORM(1)
YES
0
YES
NO
1
N
YES
N>0
NO
YES
ERROR(form, 1)
N
fact
N
1
N-1

1.10. Example (Simple CDG)



- ($\stackrel{dd}{\rightarrow}$) represents direct dependence between expressions

1.11. Example (Composed CDG)



- $\stackrel{inhdep}{\rightarrow}$ represents inherited dependence (do not fail)
- $\stackrel{resdep}{\rightarrow}$ denotes the resumption dependencies (may fail)

2. Dependence Graph

2.1. Extending the Control Dependence Graph

- The previously created composed graph is insufficient for complex analyses
- Data-flow and data dependence is also calculated

Created by XMLmind XSL-FO Converter.

- Data dependence is calculated from the data-flow graph
- The composed CDG is extended with data-flow and data dependence edges

2.2. Data Dependence

- We define data dependence between two nodes $n_1 \overset{ddep}{\rightsquigarrow} n_2$ if:
  - there is a direct dependency edge between them – $n_1 \overset{dep}{\rightarrow} n_2$
  - n_2 is reachable from n_1, so the value of n_1 can flow to n_2 – $n_1 \overset{0f}{\rightsquigarrow} n_2$
- The data dependence relation ($\overset{ddep}{\rightsquigarrow}$) is transitive:

$$\frac{n_1 \overset{ddep}{\rightsquigarrow} n_2,\ n_2 \overset{ddep}{\rightsquigarrow} n_3}{n_1 \overset{ddep}{\rightsquigarrow} n_3}$$

2.3. Further Dependencies

- The graph can be extended with behaviour dependencies
- The behaviour dependence provide information about the behavioural changes when something is modified
- The change can be a data manipulation
- The more information is involved the more accurate the dependence graph is

Created by XMLmind XSL-FO Converter.

Chapter 11. Lecture 11

1. First order data-flow

1.1. Extended example module

```
  -module(dataflow).

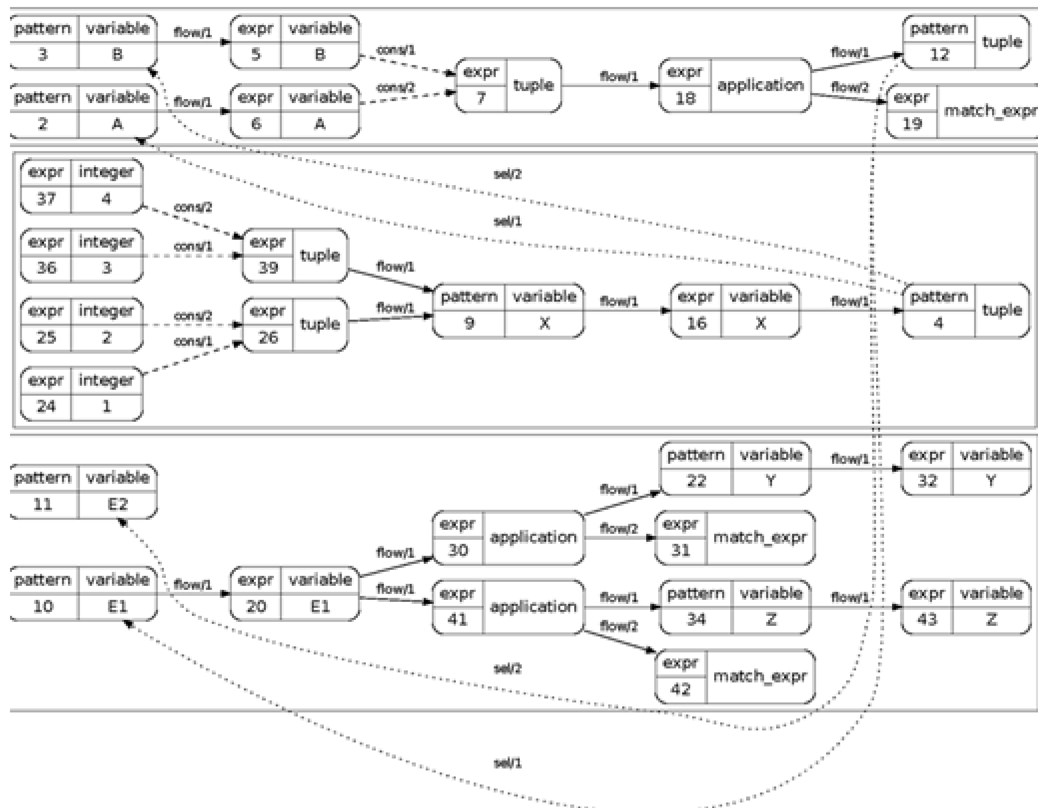
swap({A, B}) -> {B, A}.

get_1st(X) ->
    {E1, E2} = swap(X),
    E1.

const()->
    Y = get_1st({1,2}),
    Y.

const2()->
    Z = get_1st({3,4}),
    Z.
```

1.2. 0th order DFG for the extended `dataflow` module



1.3. Problems with the 0th order data-flow analysis

- Determine the value for variable `Z` (expression 43)

Created by XMLmind XSL-FO Converter.

- The possible values are *2* and *4*
- While looking into the source code, it is obvious that it can be only the value *4* (lack of the context information)

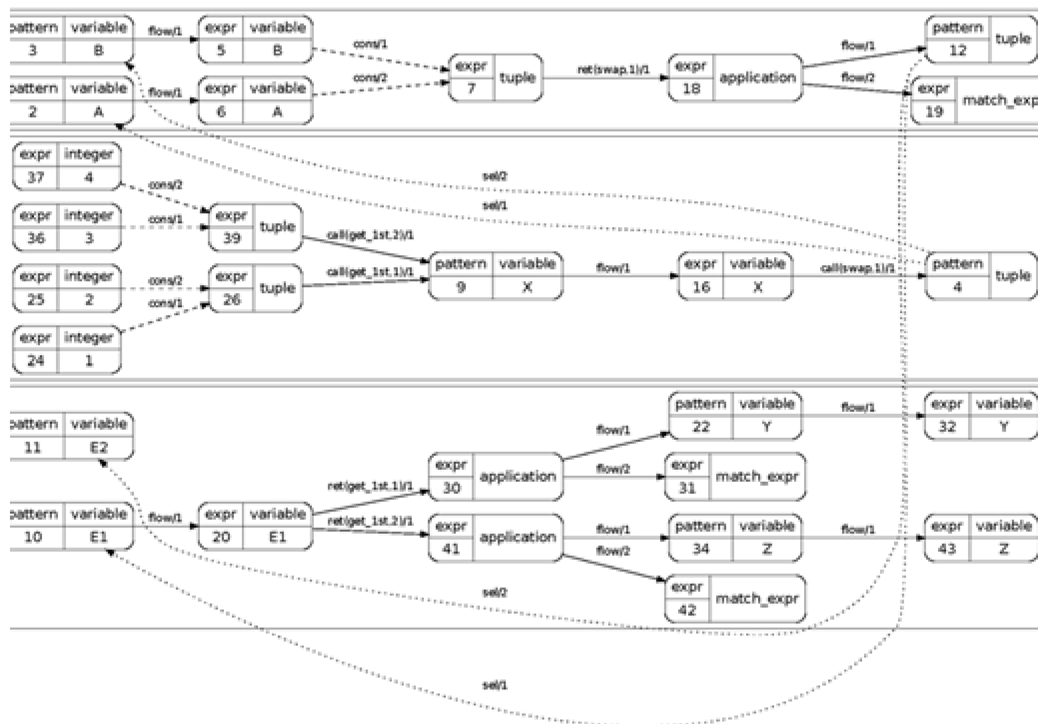
1.4. 1st order data-flow analysis

- The 0th order analysis is an over-approximation
- It does not consider the calling context of functions
- The first order data-flow analysis addresses this problem:
  - entry point with calling context information
  - return point with calling context information
- With the 1st order data-flow analysis we can avoid some false positive results

1.5. Extending the data-flow rules

- $\xrightarrow{call(g,i)}$: the ith call of the function
- $\xrightarrow{ret(g,i)}$: the return point of the ith function call

1.6. 1th order DFG for the extended `dataflow` module



1.7. Formal rule for the function call

Created by XMLmind XSL-FO Converter.

Expressions	Graph edges
: $g(e_1, \ldots, e_n)$	
'n:	$e^1_{l_1} \xrightarrow{ret(g,i)} e_0, \ldots, e^m_{l_m} \xrightarrow{ret(g,i)}$
$p^1_1, \ldots, p^1_n)$ when $g_1 \rightarrow$ $e^1_1, \ldots, e^1_{l_1}$;	$e_1 \xrightarrow{call(g,i)} p^1_1, \ldots, e_1 \xrightarrow{call(g,i}$
$\vdots$	$\vdots$
$p^m_1, \ldots, p^m_n)$ when $g_m \rightarrow$ $e^m_1, \ldots, e^m_{l_m}$.	$e_n \xrightarrow{call(g,i)} p^1_n, \ldots, e_n \xrightarrow{call(g,i}$
; the ith analysed call inction $m : g/n$	

1.8. Deriving from the 0th order data-flow rules(1)

- $\overset{0f'}{\rightsquigarrow}$ denotes the zeroth order data-flow relation calculated on the data-flow graph
- $\overset{1f[\mu]}{\rightsquigarrow}$ denotes the first order data-flow relation
- ($\overset{1f[\mu]}{\rightsquigarrow}$) μ is a list of call ($\overset{call(g,i)}{\longrightarrow}$) and return ($\overset{ret(h,j)}{\longrightarrow}$) points
- each node (n_i) that is reachable in the extended representation with the 0th order data-flow relation is reachable by the first order relation (0th *flow rule*)

1.9. Deriving from the 0th order data-flow rules(2)

- if a data constructor packs ($\overset{c_i}{\longrightarrow}$) the node n_1 into n_2 and the value of n_2 flows (with a first order flow) into the node n_3 and another data constructor unpacks ($\overset{s_i}{\longrightarrow}$) the value into n_4, then the value of n_1 flows into n_4 (1st *c-s rule*).
- the call ($\overset{call(g,i)}{\longrightarrow}$) edge behaves similarly as the flow $\overset{f}{\longrightarrow}$ edges, so the data flows through the (*call rule*
- the return ($\overset{ret(h,j)}{\longrightarrow}$) edge behave similarly as the flow $\overset{f}{\longrightarrow}$ edges, so the data flows through the *return rule*)
- the data can flow through any function call (*call concat. rule*)

1.10. Deriving from the 0th order data-flow rules(3)

- if the value of the node n_1 flows into the node n_2 through the return value of a function call and the value of n_2 flows into the node n_3 through the return value of another function call, then the value of n_1 transitively flows into the node n_3 (*return concat. rule*)
- entering the function through the edge $\overset{call(g,i)}{\longrightarrow}$, implies that we have to leave the function through the $\overset{ret(g,i)}{\longrightarrow}$ edge (*reduce rule*) and leaving the function body through an $\overset{ret(g,j)}{\longrightarrow}$ $(j \neq i)$ edge is not allowed ($Lemma\ 3$)

Created by XMLmind XSL-FO Converter.

1.11. Notations for `Definition 4`

- μ denotes a list;
- $hd(\mu)$ results the head (first) element of a list;
- $last(\mu)$ stands for the last element of a list;
- $\mu++\rho$ denotes the concatenation of list μ and list ρ;
- μ_n denotes the nth element of list μ.

1.12. Definition 4 (1)

The data-flow relation ($\overset{1f}{\leadsto}$) is the minimal relation that satisfies the following rules:

$$\frac{n_1 \overset{0f'}{\leadsto} n_2}{n_1 \overset{1f[]}{\leadsto} n_2} \qquad (0^{th} \text{ flow rule})$$

$$\frac{n_1 \overset{c_i}{\to} n_2,\ n_2 \overset{1f[\mu]}{\leadsto} n_3,\ n_3 \overset{s_i}{\to} n_4}{n_1 \overset{1f[\mu]}{\leadsto} n_4} \qquad (1^{st} \text{ c-s rule})$$

$$\frac{n_1 \overset{call(g,i)}{\to} n_2}{n_1 \overset{1f[call_{(g,i)}]}{\leadsto} n_2} \qquad (\text{call rule})$$

1.13. Definition 4 (2)

$$\frac{n_1 \overset{ret(h,j)}{\to} n_2}{n_1 \overset{1f[ret_{(h,j)}]}{\leadsto} n_2} \qquad (\text{return rule})$$

$$\frac{n_1 \overset{1f[\mu]}{\leadsto} n_2,\ n_2 \overset{1f[\rho]}{\leadsto} n_3}{n_1 \overset{1f[\mu++\rho]}{\leadsto} n_3} \quad if\ (\exists f\ \exists i:\ (hd(\rho) = call_{(g,i)}))\ or\ \rho = [] \qquad (\text{call concat. rule})$$

$$\frac{n_1 \overset{1f[\mu]}{\leadsto} n_2,\ n_2 \overset{1f[ret_{(h,j)}|\rho]}{\leadsto} n_3}{n_1 \overset{1f[\mu++[ret_{(h,j)}|\rho]]}{\leadsto} n_3} \quad if\ (\exists f\ \exists i:\ (last(\mu) = ret_{(g,i)}))\ or\ \mu = [] \qquad (\text{return concat. rule})$$

$$\frac{n_1 \overset{1f[\mu++[call_{(h,i)}]]}{\leadsto} n_2,\ n_2 \overset{1f[ret_{(h,i)}]}{\leadsto} n_3}{n_1 \overset{1f[\mu]}{\leadsto} n_3} \qquad \text{(reduce rule)}$$

2. Higher order data-flow analysis

2.1. N^{th} Order Analysis

- Generalisation of the 1st order analysis
- Calling context information in two steps
- Iteratively generalise to store the context in N steps

2.2. Why generalisation is required?

Consider the following example:

```
func(Fun, Data)->
    Fun(Data).

call_pear()->
    func(fun pear/1, [pear]).

call_apple()->
    func(fun apple/1, [apple]).
```

2.3. Why generalisation is required?

- In the 1st order data-flow graph:
  - From `fun pear/1` expression $\overset{call(pear,i)}{\rightarrow}$ to the `Fun` argument of function `func/2`
  - From `fun pear/1` expression $\overset{call(apple,j)}{\rightarrow}$ to the `Fun` argument of function `func/2`
  - From the argument `Fun` a flow edge to the `Fun(Data)` application
  - From the `Fun(Data)` application two edges to the corresponding function call: $\overset{ret(pear,i)}{\rightarrow}$, $\overset{ret(apple,j)}{\rightarrow}$
- The first order reaching from the body of $call_pear/0$ results in that both function $apple/1$ and function $pear/1$ were called
- Possible solution is 2nd order analysis: $\overset{call((func,1);(pear,i))}{\rightarrow}$, $\overset{call((func,2);(apple,j))}{\rightarrow}$

Created by XMLmind XSL-FO Converter.

Chapter 12. Lecture 12

1. Concurrent data-flow

1.1. Message passing

- Another way for transferring data
- Approximation of data-flow (conservative)
- Similar to zeroth order data-flow trough function calls
- Each send message linked to receive sides
  - huge DFG with large amount of improper flow edges
  - need to be restricted (context information)

1.2. Processes and Message Passing

- Built in concurrency
- Remote process spawning
- Processes at virtual machine level
- Light-weight processes
- Communication through message passing
- Message queue
- Basic language constructs for concurrency: spawn, register, send and receive

1.3. Concurrent data-flow analysis

- The analysis can be easily extended for distributed programs
- Concurrency (reminder):
  - Spawning processes
  - Registering processes
  - Sending messages
  - Receiving

1.4. Detecting Spawned Processes

- Statically detecting the recipient is not straightforward
- In some cases it is impossible to calculate
- Difficulties with processes
  - Process identifiers (PIDs) are created dynamically
  - PIDs can be passed as parameters

Created by XMLmind XSL-FO Converter.

1.5. Example

```
send_data(Pid) ->
  Data = do_some_computation(),
  Pid  ! {"Sending computed data", Data}.
```

- Do we have information about `Pid`?
- Do we know where the message is sent?

1.6. Process analysis

- Detect process identifiers: PIds and registered names
- Analyse functions that can be potentially spawned with first order data-flow reaching
- Functions are identified with triples:

  `(ModName, FunName, length(ArgList))`

- Reminder:
  - `spawn/1,2,3,4`
  - `spawn_link/1,2,3,4`
  - `spawn_monitor/1,3`
  - etc.

1.7. Function Analysis

The following sets define the possible values of elements in the triple:

- $MN = \{n \in V \mid n \overset{1f}{\rightsquigarrow} mn, \nexists n', n' \in V, n' \neq n : n' \overset{1f}{\rightsquigarrow} n\}$
- $FN \ = \{n \in V \mid n \overset{1f}{\rightsquigarrow} fn, \nexists n', n' \in V, n' \neq n : n' \overset{1f}{\rightsquigarrow} n\}$
- $Arg \ = \{n \in V \mid n \overset{1f}{\rightsquigarrow} arg, \nexists n', n' \in V, n' \neq n : n' \overset{1f}{\rightsquigarrow} n\}$
- $SF_{Pid} = \{(val(M), val(F), size(A)) \mid M \in MN, F \in FN, A \in Arg\}$

Where:

- $mn \in V$ – node representing the module name
- $fn \in V$ – node representing the function name
- $arg \in V$ – node representing the argument list in the Data-Flow Graph – $DFG = (V, E)$

1.8. Example(1)

```
  -module(mymod).

start(Fun, Args) ->
   Pid = spawn(?MODULE, Fun, Args).
   Pid ! start,
   Pid.
```


Created by XMLmind XSL-FO Converter.

```
init() -> start(loop1, [init, []]).

process(Data) -> start(loop2, [proc, Data]).

loop1(State, Data) ->
   ...
loop2(Tag, Data) ->
   ...
```

1.9. Example(2)

Sets:

- $MN = \{\$?MODULE\$\}$
- $FN = \{\$loop1\$, \$loop2\$\}$
- $Arg = \{\$[init, []]\$, \$[proc, Data]\$\}$
- $SF_{Pid} = \{\{mymod, loop1, 2\}, \{mymod, loop2, 2\}\}$

1.10. Detecting registered processes

- Messages can be sent to registered processes
- The alias is an atom
- Reminder: `register(Name, PId)`
- Identify the aliases and possibly related processes:
  - Values of `Name`
  - Potential processes passed as `PId`
- Backward data-flow reaching

1.11. Calculating values for a *register* call

- $AN = \{n \in V \mid n \overset{1f}{\rightsquigarrow} an, \nexists n', n' \in V, n' \neq n : n' \overset{1f}{\rightsquigarrow} n\}$
- $Atom_{AN} = \{n \in V \mid \exists n',\ n' \in AN,\ \exists n'',\ n'' \in V,$

  $n'' \overset{1f}{\rightsquigarrow} n,\ val(n'') = val(n'),\ type(n') = atom,\ n \in MPass\}$
- $PN = \{n \in V \mid n \overset{1f}{\rightsquigarrow} pn, type(n) = spawn_app\}$

where

- an – node representing the `Alias`
- pn – node representing the process identifier (`PidExpr`)
- $MPass$ – elements of the message passing expression
- $type(n) = spawn_app$ – node n is an application of function $spawn_app$

1.12. Calculating possible functions

- Set PM denotes the function application nodes calling the function $spawn*/n$
- Calculate the SF_{Pid_i} set, for every element of the $Pid_i \in PM$ set
- SF_{Alias} is the union of these sets, since all of these functions are potentially registered with examined $Alias$
- $\forall A \in Atom_{AN} : SF_A = SF_{Alias}$

1.13. Modified example(1)

```
   start(Fun, Args) ->
    Pid = spawn(?MODULE, Fun, Args).
    Pid ! start,
    Pid.

 init(Alias) ->
    P = start(loop1, [init, []]),
    register(Alias, P).

 loop1(State, Data) -> ...

 reg_proc() ->
    init(proc1),
    proc1 ! some_message.
```

1.14. Modified example(2)

- $AN = \{\$proc1\$\}$

  – from function call `init(proc1)`

- $Atom_{AN} = \{\$proc1\$\}$

  – from expression `proc1 ! some_msg`

- $PN = \{\$spawn(?MODULE, Fun, Args)\$\}$
- $SF_{Alias} = SF_{Pid} = \{\{mymod, loop1, 2\}\}$
- $SF_{proc1} = SF_{Alias} = \{\{mymod, loop1, 2\}\}$

1.15. Heuristics

- Ideally MN, FN and Arg sets contain only atoms
- For industrial sized source code it is not the case
- Approximations of concurrent data-flow
- Lack of some details
- Example: variables representing the modules, functions and argument lists can be only estimated (statically calculated dynamic information)

1.16. Heuristics based on the partial knowledge (1)

Created by XMLmind XSL-FO Converter.

- The name of the module (m) and the name of the function (f) are atoms and the arity (a) is unknown – we select all functions with the name f from the module without regarding its arity a_i and we add $\{m, f, a_i\}$ to SF_*;
- The name of the module (m) is an atom and the function name f and arity (a) is unknown – we select all functions from the module without regarding its name f_i and its arity a_i and we add $\{m, f_i, a_i\}$ to SF_* for each function f_i/a_i;

1.17. Heuristics based on the partial knowledge (2)

- The name of the module (m) is an atom and we can calculate the length of the parameter list (a) and the function (f) is unknown – we select all functions f_i from the module with the calculated arity a and we add $\{m, f_i, a\}$ to SF_*;
- The name of the function (f) is an atom and we can calculate the length of the parameter list (a) and the module (m) is unknown – we select every module (m_i) that defines a function `f/n` and we add $\{m_i, f, a\}$ to SF_*.

1.18. Example (1)

```
  start(Fun, Args) ->
   Pid = spawn(?MODULE, Fun, Args).
   Pid ! start,
   Pid.

init(Alias, FunName) ->
   P = start(FunName, [init, []]),
   register(Alias, P).

loop1(State, Data) ->
   ...
loop2(Tag, Data) ->
   ...
```

1.19. Example (2)

- $MN = \{\$?MODULE\$\}$
- $FN = \{\$FunName\$\}$
- $Arg = \{\$[init, []]\$, \$[proc, Data]\$\}$
- $SF_{Pid} = \{\{mymod, loop1, 2\}, \{mymod, loop2, 2\}\}$

1.20. Possible message recipient at sender side (1)

- Reminder: (`!`): $e_1 \; ! \; e_2$, (`send/2`): $send(e1, e2)$
- Identifying the recipient (e_1) by analysing the send expressions
- Backward data-flow from the e_1 expression to identify *spawn*s and *register*s
- Calculated sets:
  - $Spawn = \{n \in V | n \overset{1f}{\rightsquigarrow} e_1, type(n) = spawn_app\}$

Created by XMLmind XSL-FO Converter.

- $Reg = \{n \in V | n \overset{1f}{\rightsquigarrow} e_1, type(sup(n)) = reg_app\}$
- $sup(n)$ is the superior expression of node n

- By examining the sets the previous heuristics can be applied

1.21. Possible message recipient at sender side (2)

- When the expression e_1 is an atom, or the backward reaching returns an atom:
  - Reaching is not suitable for determining the set SF_{e_1}, because there is not data-flow connection between them (the recipient is a registered process)
  - We need to calculate the sets AN and SF_A first ($A \in Atom_{AN}$)
  - $SF_{e_1} = \bigcup_{i=1}^{k} SF_{name_i}$, where $\forall i \in [1..k] : name_i$ is a possible value of expression e_i

1.22. Analysis of receivers

- Executed functions are analysed
- Transitive closure of the call chain is also analysed
- $Rec = \{n \in V \mid type(n) = receive_expr,\ F \in tr_closure(SF_{e_1}),\ n \in body(F)\}$
- Where:
  - *tr_closure* returns the transitive closure of the $\overset{call}{\rightarrow}$ relation
  - $f_1 \overset{call}{\rightarrow} f_2$ means that function f_1 calls function f_2 and f_i is represented by the previously defined triple
  - `body/1` returns the expressions from the body of a function

1.23. Concurrent data-flow rule

Expression: Edges: e_0:

$e_1 \ ! \ e_2$

e':

receive

$p_1 \ when \ g_1 \rightarrow$

$e_1^1, \ldots, e_{l_1}^1;$

$\vdots$

$p_n \ when \ g_n \rightarrow$

$e_1^n, \ldots, e_{l_n}^n$

after

$e \rightarrow e_1, \ldots, e_s$

Created by XMLmind XSL-FO Converter.

end

$e_2 \xrightarrow{f} e_0$

$e_2 \xrightarrow{f} p_1, \ldots, e_2 \xrightarrow{f} p_n$

$e^1_{l_1} \xrightarrow{f} e', \ldots, e^n_{l_n} \xrightarrow{f} e'$

$e_s \xrightarrow{f} e'$

1.24. Connection between send and receive sides

- We apply the data-flow rule for every $e' \in Rec_{e_1}$
- The sent message e_2 flows into the different patterns ($p_1, \ldots, p_n$) of the selected receive expression (i.e. $e_2 \xrightarrow{f} p_i$)
- The result of the receive expression can be the value of the last expression of its clauses ($e^i_j \xrightarrow{f} e'$)
- The result of the send expression is the message itself ($e_2 \xrightarrow{f} e_0$)

1.25. Extending the previous example (1)

```
loop1(State, Data) ->
 receive
   start ->
     Data = initial_steps(),
     loop(started, Data);
   Msg ->
     NewData = process_msg(Msg, Data),
     loop(State, NewData);
   stop ->
     closing_steps()
 end.
```

1.26. Extending the previous example (2)

In the example there were two send expressions:

- `Pid ! start`
  - $Spawn = \{\$spawn(?MODULE, Fun, Args)\$\}$
  - $SF_{Pid} = \{\{mymod, loop1, 2\}\}$
  - The only receive expression is in the body of the function `mymod:loop1/2`
  - Link the sent message to its patterns: $\$start\$ \xrightarrow{f} p$, where $p \in \{\$start\$, \$Msg\$, \$stop\$\}$
- `proc1 ! some_message`
  - $SF_{proc1} = \{\{mymod, loop1, 2\}\}$
  - The only receive expression is in the body of the function `mymod:loop1/2`

Created by XMLmind XSL-FO Converter.

- Link the message and the patterns in the Data-Flow Graph: $\$some_message\$ \xrightarrow{f} p_i$.

1.27. Refining the analysis

- The presented analysis is an overestimation
- The analysis does not consider the order of the messages and liveness of the processes
- The analysis disregards the fact of unregistering processes
- Detect liveness (Control-flow – extending the analysis with possible execution paths)
- Consider $exit$ signals

1.27.1. Improving the 1st Order Data-Flow Analysis

1.28. Refining the 1st Order Data-Flow Analysis

- Split the DFG building algorithm into two parts
  1. Calculating the sequential DFG
  2. Calculating the concurrent data-flow edges
- Running the process analysis iteratively until it reaches its fixed point results in a more accurate graph
- The iteration terminates when there are no more new message passing expression or every receive side is connected to sending expressions

1.29. Example (1)

```
   start() ->
    Pid1 = spawn(?MODULE, fun1, []),
    Pid2 = spawn(?MODULE, fun2, []).
    Pid1 ! {pid, Pid2}.

 fun1() ->
    receive
        {pid, Pid} -> Pid ! some_message
    end.

 fun2(Tag, Data) ->
    receive
        A -> do_sth(A)
    end.
```

1.30. Example (2)

- The backward reaching on the sequential data-flow graph will not find the origin of `Pid` in the message passing
- That results, that we can not deduce that it refers to function `fun2/0`
- In the second stage a new flow edge is inserted that connects the sent message and the receive pattern in `fun1/0`: $\$\{pid, Pid1\}\$ \xrightarrow{f} \$\{pid, Pid\}\$$
- Performing backward reaching we get the origin of `Pid` and we can deduce that it refers to `fun2/0`

Created by XMLmind XSL-FO Converter.

- New flow edge: $\$some_message\$ \xrightarrow{f} \$A\$$.

Created by XMLmind XSL-FO Converter.

Chapter 13. Lecture 13

1. Considered language elements

1.1. Examined Language Constructs

- `Pid = spawn(Mod, Fun, Args)`
- `register(Name, Pid)`
- `Pid ! Message`
- `Name ! Message`
-

```
 receive
Pattern1 -> do_sth;
Pattern2 -> do_sth_else;
...
end
```

1.2. ETS tables

- Erlang Term Storage
- “Shared memory”
- Interfaced by library functions: `new/2, insert/2, match/2, select/2`

2. Communication Model

2.1. Representation

- Directed, rooted, labelled graph
- Nodes: processes (`m:f/a`)
- Edges:
  - $spawn$, $register$
  - $\{send, Message\}$
  - $create$, $\{read, Pattern\}$, $\{write, Data\}$
- Root node: SP – the “super process”

2.2. Non trivial steps...

1. Process identification
2. Adding process communication edges

Created by XMLmind XSL-FO Converter.

3.

Adding hidden communication edges

2.3. The Magic Behind the Steps

- Data-Flow Analysis
- Technique for gathering information about the possible set of values calculated at various points in a program
- Data Flow Graph - DFG containing the direct relations
- Data-Flow Reaching to calculate indirect flow: $n_1 \overset{1f}{\rightsquigarrow} n_2$

3. Motivating Example

```
 connect(Cli) ->
    ?Name ! {connect, Cli}.

do(Mod, Fun, Tab)->
    ?Name ! {do, Mod, Fun, Tab}.

start() ->
    register(?Name, spawn_link(?MODULE, init, [])).

init()->
    ?MODULE:loop([]).

loop(State)->
    receive
        {connect, Cli} ->
            ?MODULE:loop([Cli|State]);
        {do, Mod, Fun, Tab} ->
            handle_job(Mod, Fun, Tab),
            ?MODULE:loop(State)
    end.

handle_job(Mod, Fun, Tab) ->
    Data = ets:select(Tab, [{{'$1','$2'}, [{'/=', '$1', result}], ['$$']}]),
    Result = Mod:Fun(Data),
    ets:insert(Tab, {result, Result}).



start(Client) ->
    server:connect(Client),
    ets:new(data, [named_table, public]),
    spawn(?MODULE, input, [self()]),
    loop(data, Client).

loop(Tab, Name) ->
    receive
        {job, {Mod, Fun}} ->
            server:do(Mod, Fun, Tab),
            loop(Tab, Name)
    end.

input(Loop) ->
    case read_input() of
        Job ->
            Loop ! {job, Job},
            input(Loop)
    end.

read_input() ->
    [ets:insert(data, Data) || Data <- init_data()],
```


Created by XMLmind XSL-FO Converter.

```
      returns_the_job_to_be_executed().
```

4. Algorithm

4.1. Process Identification

1.

Add a process node for each `spawn*` and link it to its parent process

2.

Add a process node for each function which takes part in communication:

- Belongs to a spawned process
- Belongs to an interface (linked to SP)

3.

Add process registration information

```
 connect(Cli) ->
    ?Name ! {connect, Cli}.

do(Mod, Fun, Tab)->
    ?Name ! {do, Mod, Fun, Tab}.

start() ->
    register(?Name, spawn_link(?MODULE, init, [])).

init()->
    ?MODULE:loop([]).

loop(State)->
    receive
        {connect, Cli} ->
            ?MODULE:loop([Cli|State]);
        {do, Mod, Fun, Tab} ->
            handle_job(Mod, Fun, Tab),
            ?MODULE:loop(State)
    end.

handle_job(Mod, Fun, Tab) ->
    Data = ets:select(Tab, [{{'$1','$2'}, [{'/=', '$1', result}], ['$$']}]),
    Result = Mod:Fun(Data),
    ets:insert(Tab, {result, Result}).



connect(Cli) ->
    ?Name ! {connect, Cli}.

do(Mod, Fun, Tab)->
    ?Name ! {do, Mod, Fun, Tab}.

{start()} ->
    register(?Name, spawn_link(?MODULE, init, [])).

init()->
    ?MODULE:loop([]).

loop(State)->
    receive
        {connect, Cli} ->
```

```
                ?MODULE:loop([Cli|State]);
            {do, Mod, Fun, Tab} ->
                handle_job(Mod, Fun, Tab),
                ?MODULE:loop(State)
        end.

    handle_job(Mod, Fun, Tab) ->
        Data = ets:select(Tab, [{{'$1','$2'}, [{'/=', '$1', result}], ['$$']}]),
        Result = Mod:Fun(Data),
        ets:insert(Tab, {result, Result}).



    connect(Cli) ->
        ?Name ! {connect, Cli}.

    do(Mod, Fun, Tab)->
        ?Name ! {do, Mod, Fun, Tab}.

    {start() } ->
        register(?Name, spawn_link(?MODULE, init, [])).

    {init()->
        ?MODULE:loop([]).}

    loop(State)->
        receive
            {connect, Cli} ->
                ?MODULE:loop([Cli|State]);
            {do, Mod, Fun, Tab} ->
                handle_job(Mod, Fun, Tab),
                ?MODULE:loop(State)
        end.

    handle_job(Mod, Fun, Tab) ->
        Data = ets:select(Tab, [{{'$1','$2'}, [{'/=', '$1', result}], ['$$']}]),
        Result = Mod:Fun(Data),
        ets:insert(Tab, {result, Result}).



    {start(Client)} ->
        server:connect(Client),
        ets:new(data, [named_table, public]),
        {spawn(?MODULE, input, [self()])},
        loop(data, Client).

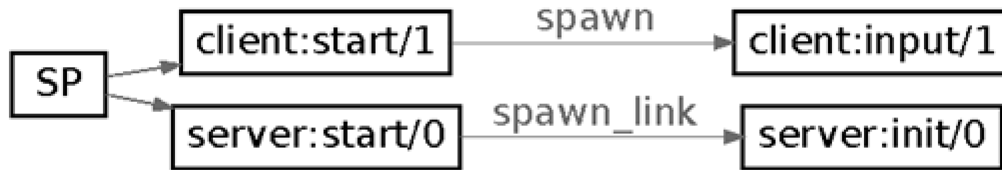
    loop(Tab, Name) ->
        receive
            {job, {Mod, Fun}} ->
                server:do(Mod, Fun, Tab),
                loop(Tab, Name)
        end.

    {input(Loop) ->
        case read_input() of
            Job ->
                Loop ! {job, Job},
                input(Loop)
        end.}

    read_input() ->
        [ets:insert(data, Data) || Data <- init_data()],
        returns_the_job_to_be_executed().
```

4.2. Process Nodes

Created by XMLmind XSL-FO Converter.




```
 connect(Cli) ->
    ?Name ! {connect, Cli}.

do(Mod, Fun, Tab)->
    ?Name ! {do, Mod, Fun, Tab}.

start() ->
    register({?Name}, spawn_link(?MODULE, init, [])).

init()->
    ?MODULE:loop([]).

loop(State)->
    receive
        {connect, Cli} ->
            ?MODULE:loop([Cli|State]);
        {do, Mod, Fun, Tab} ->
            handle_job(Mod, Fun, Tab),
            ?MODULE:loop(State)
    end.

handle_job(Mod, Fun, Tab) ->
    Data = ets:select(Tab, [{{'$1','$2'}, [{'/=', '$1', result}], ['$$']}]),
    Result = Mod:Fun(Data),
    ets:insert(Tab, {result, Result}).



connect(Cli) ->
    ?Name ! {connect, Cli}.

do(Mod, Fun, Tab)->
    ?Name ! {do, Mod, Fun, Tab}.
{-define(Name, job_server).}
start() ->
    register({?Name}, spawn_link(?MODULE, init, [])).

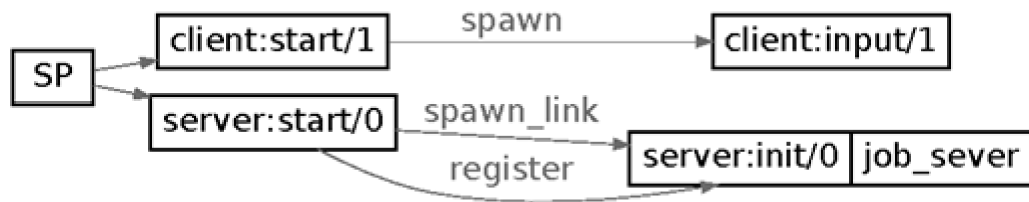
init()->
    ?MODULE:loop([]).

loop(State)->
    receive
        {connect, Cli} ->
            ?MODULE:loop([Cli|State]);
        {do, Mod, Fun, Tab} ->
            handle_job(Mod, Fun, Tab),
            ?MODULE:loop(State)
    end.

handle_job(Mod, Fun, Tab) ->
    Data = ets:select(Tab, [{{'$1','$2'}, [{'/=', '$1', result}], ['$$']}]),
    Result = Mod:Fun(Data),
    ets:insert(Tab, {result, Result}).
```

4.3. Process Nodes

Created by XMLmind XSL-FO Converter.




```
 connect(Cli) ->
    ?Name ! {connect, Cli}.

do(Mod, Fun, Tab)->
    ?Name ! {do, Mod, Fun, Tab}.

start() ->
    register(?Name, spawn_link(?MODULE, init, [])).

init()->
    ?MODULE:loop([]).

loop(State)->
    receive
        {connect, Cli} ->
            ?MODULE:loop([Cli|State]);
        {do, Mod, Fun, Tab} ->
            handle_job(Mod, Fun, Tab),
            ?MODULE:loop(State)
    end.

handle_job(Mod, Fun, Tab) ->
    Data = ets:select(Tab, [{{'$1','$2'}, [{'/=', '$1', result}], ['$$']}]),
    Result = Mod:Fun(Data),
    ets:insert(Tab, {result, Result}).



start(Client) ->
    server:connect(Client),
    ets:new(data, [named_table, public]),
    spawn(?MODULE, input, [self()]),
    loop(data, Client).

loop(Tab, Name) ->
    receive
        {job, {Mod, Fun}} ->
            server:do(Mod, Fun, Tab),
            loop(Tab, Name)
    end.

input(Loop) ->
    case read_input() of
        Job ->
            Loop ! {job, Job},
            input(Loop)
    end.

read_input() ->
    [ets:insert(data, Data) || Data <- init_data()],
    returns_the_job_to_be_executed().



connect(Cli) ->
    ?Name ! {connect, Cli}.
```


Created by XMLmind XSL-FO Converter.

```
do(Mod, Fun, Tab)->
    ?Name ! {do, Mod, Fun, Tab}.

start() ->
    register(?Name, spawn_link(?MODULE, init, [])).

init()->
    ?MODULE:loop([]).

loop(State)->
    receive
        {connect, Cli} ->
            ?MODULE:loop([Cli|State]);
        {do, Mod, Fun, Tab} ->
            handle_job(Mod, Fun, Tab),
            ?MODULE:loop(State)
    end.

handle_job(Mod, Fun, Tab) ->
    Data = ets:select(Tab, [{{'$1','$2'}, [{'/=', '$1', result}], ['$$']}]),
    Result = Mod:Fun(Data),
    ets:insert(Tab, {result, Result}).



connect(Cli) ->
    ?Name ! {connect, Cli}.

do(Mod, Fun, Tab)->
    ?Name ! {do, Mod, Fun, Tab}.

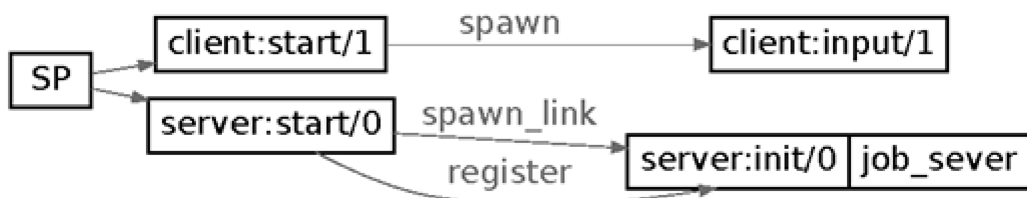
start() ->
    register(?Name, spawn_link(?MODULE, init, [])).

init()->
    {?MODULE:loop([]).}

loop(State)->
    receive
        {connect, Cli} ->
            ?MODULE:loop([Cli|State]);
        {do, Mod, Fun, Tab} ->
            handle_job(Mod, Fun, Tab),
            ?MODULE:loop(State)
    end.

handle_job(Mod, Fun, Tab) ->
    Data = ets:select(Tab, [{{'$1','$2'}, [{'/=', '$1', result}], ['$$']}]),
    Result = Mod:Fun(Data),
    ets:insert(Tab, {result, Result}).
```

4.4. Process Nodes



4.5. Process Communication

Created by XMLmind XSL-FO Converter.

- Collect message sending expressions
- Collect the appropriate receive expressions
- Calculate the send and receive containing processes
- Add edges $\{send, Message\}$ to the graph

```
 start(Client) ->
    server:connect(Client),
    ets:new(data, [named_table, public]),
    spawn(?MODULE, input, [self()]),
    loop(data, Client).

loop(Tab, Name) ->
    receive
        {job, {Mod, Fun}} ->
            server:do(Mod, Fun, Tab),
            loop(Tab, Name)
    end.

input(Loop) ->
    case read\_input() of
        Job ->
            Loop ! {job, Job},
            input(Loop)
    end.

read_input() ->
    [ets:insert(data, Data) || Data <- init_data()],
    returns_the_job_to_be_executed().



start(Client) ->
    server:connect(Client),
    ets:new(data, [named_table, public]),
    spawn(?MODULE, input, [self()]),
    loop(data, Client).

loop(Tab, Name) ->
    receive
        {job, {Mod, Fun}} ->
            server:do(Mod, Fun, Tab),
            loop(Tab, Name)
    end.

input(Loop) ->
    case read_input() of
        Job ->
            {Loop} ! {job, Job},
            input(Loop)
    end.

read_input() ->
    [ets:insert(data, Data) || Data <- init_data()],
    returns_the_job_to_be_executed().



start(Client) ->
    server:connect(Client),
    ets:new(data, [named_table, public]),
    spawn(?MODULE, input, [self()]),
    loop(data, Client).

loop(Tab, Name) ->
    receive
        {job, {Mod, Fun}} ->
```


Created by XMLmind XSL-FO Converter.

```
            server:do(Mod, Fun, Tab),
            loop(Tab, Name)
    end.

input(Loop) ->
    case read_input() of
        Job ->
            {\bf Loop} ! {job, Job},
            input(Loop)
    end.

read_input() ->
    [ets:insert(data, Data) || Data <- init_data()],
    returns_the_job_to_be_executed().



start(Client) ->
    server:connect(Client),
    ets:new(data, [named_table, public]),
    spawn(?MODULE, input, [self()]),
    loop(data, Client).

loop(Tab, Name) ->
    receive
        {job, {Mod, Fun}} ->
            server:do(Mod, Fun, Tab),
            loop(Tab, Name)
    end.

input(Loop) ->
    case read_input() of
        Job ->
            {\bf Loop} ! {job, Job},
            input(Loop)
    end.

read_input() ->
    [ets:insert(data, Data) || Data <- init_data()],
    returns_the_job_to_be_executed().



{start(Client) ->
    server:connect(Client),
    ets:new(data, [named_table, public]),
    spawn(?MODULE, input, [self()]),
    loop(data, Client).}

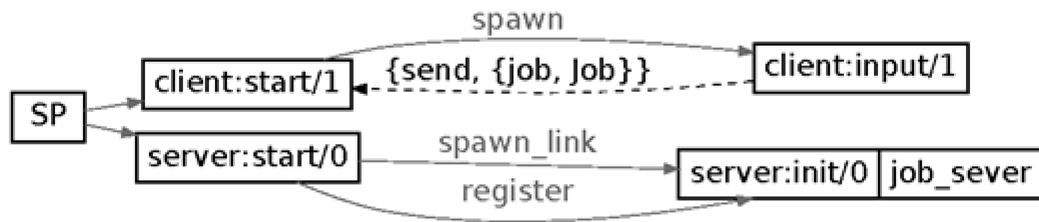
loop(Tab, Name) ->
    receive
        {job, {Mod, Fun}} ->
            server:do(Mod, Fun, Tab),
            loop(Tab, Name)
    end.

{input(Loop) ->
    case read\_input() of
        Job ->
            {Loop} ! {job, Job},
            input(Loop)
    end.}

read_input() ->
    [ets:insert(data, Data) || Data <- init_data()],
    returns_the_job_to_be_executed().
```

4.6. Process Communication

Created by XMLmind XSL-FO Converter.




```
 connect(Cli) ->
    ?Name ! {connect, Cli}.

do(Mod, Fun, Tab)->
    ?Name ! {do, Mod, Fun, Tab}.

start() ->
    register(?Name, spawn_link(?MODULE, init, [])).

init()->
    ?MODULE:loop([]).

loop(State)->
    receive
        {connect, Cli} ->
            ?MODULE:loop([Cli|State]);
        {do, Mod, Fun, Tab} ->
            handle_job(Mod, Fun, Tab),
            ?MODULE:loop(State)
    end.

handle_job(Mod, Fun, Tab) ->
    Data = ets:select(Tab, [{{'$1','$2'}, [{'/=', '$1', result}], ['$$']}]),
    Result = Mod:Fun(Data),
    ets:insert(Tab, {result, Result}).



start(Client) ->
    server:connect(Client),
    ets:new(data, [named_table, public]),
    spawn(?MODULE, input, [self()]),
    loop(data, Client).

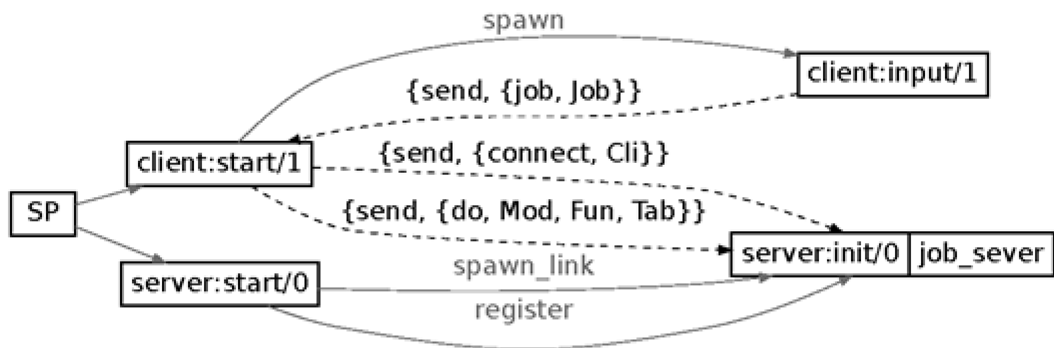
loop(Tab, Name) ->
    receive
        {job, {Mod, Fun}} ->
            server:do(Mod, Fun, Tab),
            loop(Tab, Name)
    end.

input(Loop) ->
    case read_input() of
        Job ->
            Loop ! {job, Job},
            input(Loop)
    end.

read_input() ->
    [ets:insert(data, Data) || Data <- init_data()],
    returns_the_job_to_be_executed().
```

4.7. Process Communication

Created by XMLmind XSL-FO Converter.



4.8. Hidden Communication

- Create a process node for each created ETS table and link it to its parent
- Detect named ETS table
- Collect write and read ETS operations and add them to the graph

```
 start(Client) ->
    server:connect(Client),
    ets:new(data, [named_table, public]),
    spawn(?MODULE, input, [self()]),
    loop(data, Client).

loop(Tab, Name) ->
    receive
        {job, {Mod, Fun}} ->
            server:do(Mod, Fun, Tab),
            loop(Tab, Name)
    end.

input(Loop) ->
    case read_input() of
        Job ->
            Loop ! {job, Job},
            input(Loop)
    end.

read_input() ->
    [ets:insert(data, Data) || Data <- init\_data()],
    returns_the_job_to_be_executed().



{start(Client) ->
    server:connect(Client),
    ets:new({data}, [{named_table}, public]),
    spawn(?MODULE, input, [self()]),
    loop(data, Client).}

loop(Tab, Name) ->
    receive
        {job, {Mod, Fun}} ->
            server:do(Mod, Fun, Tab),
            loop(Tab, Name)
    end.

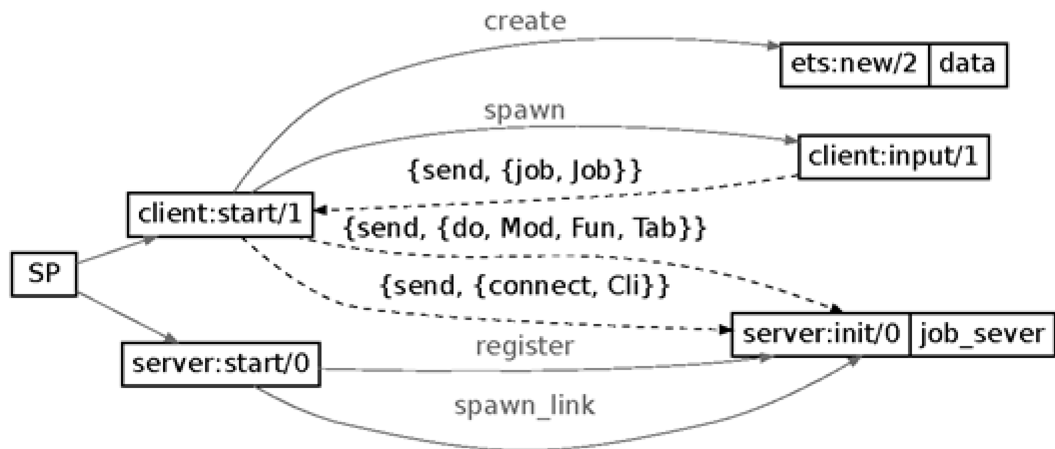
input(Loop) ->
    case read_input() of
        Job ->
            Loop ! {job, Job},
```


Created by XMLmind XSL-FO Converter.

```
            input(Loop)
    end.

read_input() ->
    [ets:insert(data, Data) || Data <- init_data()],
    returns_the_job_to_be_executed().
```

4.9. Hidden Process Nodes




```
 connect(Cli) ->
    ?Name ! {connect, Cli}.

do(Mod, Fun, Tab)->
    ?Name ! {do, Mod, Fun, Tab}.

start() ->
    register(?Name, spawn_link(?MODULE, init, [])).

init()->
    ?MODULE:loop([]).

loop(State)->
    receive
        {connect, Cli} ->
            ?MODULE:loop([Cli|State]);
        {do, Mod, Fun, Tab} ->
            handle_job(Mod, Fun, Tab),
            ?MODULE:loop(State)
    end.

handle_job(Mod, Fun, Tab) ->
    Data = ets:select(Tab, [{{'$1','$2'}, [{'/=', '$1', result}], ['$$']}]),
    Result = Mod:Fun(Data),
    ets:insert({Tab}, {result, Result}).



connect(Cli) ->
    ?Name ! {connect, Cli}.

do(Mod, Fun, Tab)->
    ?Name ! {do, Mod, Fun, Tab}.

start() ->
    register(?Name, spawn_link(?MODULE, init, [])).
```


Created by XMLmind XSL-FO Converter.

```
init()->
    ?MODULE:loop([]).

loop(State)->
    receive
        {connect, Cli} ->
            ?MODULE:loop([Cli|State]);
        {do, Mod, Fun, Tab} ->
            handle_job(Mod, Fun, Tab),
            ?MODULE:loop(State)
    end.

handle_job(Mod, Fun, Tab) ->
    Data = ets:select(Tab, [{{’$1’,’$2’}, [{’/=’, ’$1’, result}], [’$$’]}]),
    Result = Mod:Fun(Data),
    ets:insert({Tab}, {result, Result}).



connect(Cli) ->
    ?Name ! {connect, Cli}.

do(Mod, Fun, Tab)->
    ?Name ! {do, Mod, Fun, Tab}.

start() ->
    register(?Name, spawn_link(?MODULE, init, [])).

init()->
    ?MODULE:loop([]).

loop(State)->
    receive
        {connect, Cli} ->
            ?MODULE:loop([Cli|State]);
        {do, Mod, Fun, Tab} ->
            handle_job(Mod, Fun, Tab),
            ?MODULE:loop(State)
    end.

handle_job(Mod, Fun, Tab) ->
    Data = ets:select(Tab, [{{’$1’,’$2’}, [{’/=’, ’$1’, result}], [’$$’]}]),
    Result = Mod:Fun(Data),
    ets:insert({Tab}, {result, Result}).



connect(Cli) ->
    ?Name ! {connect, Cli}.

do(Mod, Fun, Tab)->
    ?Name ! {do, Mod, Fun, Tab}.

start() ->
    register(?Name, spawn_link(?MODULE, init, [])).

init()->
    ?MODULE:loop([]).

loop(State)->
    receive
        {connect, Cli} ->
            ?MODULE:loop([Cli|State]);
        {do, Mod, Fun, Tab} ->
            handle_job(Mod, Fun, Tab),
            ?MODULE:loop(State)
    end.

handle_job(Mod, Fun, Tab) ->
    Data = ets:select(Tab, [{{’$1’,’$2’}, [{’/=’, ’$1’, result}], [’$$’]}]),
    Result = Mod:Fun(Data),
```


Created by XMLmind XSL-FO Converter.

```
    ets:insert({Tab}, {result, Result}).



connect(Cli) ->
    ?Name ! {connect, Cli}.

do(Mod, Fun, Tab)->
    ?Name ! {do, Mod, Fun, Tab}.

start() ->
    register(?Name, spawn_link(?MODULE, init, [])).

init()->
    ?MODULE:loop([]).

loop(State)->
    receive
        {connect, Cli} ->
            ?MODULE:loop([Cli|State]);
        {do, Mod, Fun, Tab} ->
            handle_job(Mod, Fun, Tab),
            ?MODULE:loop(State)
    end.

handle_job(Mod, Fun, Tab) ->
    Data = ets:select(Tab, [{{’$1’,’$2’}, [{’/=’, ’$1’, result}], [’$$’]}]),
    Result = Mod:Fun(Data),
    ets:insert({Tab}, {result, Result}).



connect(Cli) ->
    ?Name ! {connect, Cli}.

do(Mod, Fun, Tab)->
    ?Name ! {do, Mod, Fun, Tab}.

start() ->
    register(?Name, spawn_link(?MODULE, init, [])).

init()->
    ?MODULE:loop([]).

loop(State)->
    receive
        {connect, Cli} ->
            ?MODULE:loop([Cli|State]);
        {do, Mod, Fun, Tab} ->
            handle_job(Mod, Fun, Tab),
            ?MODULE:loop(State)
    end.

handle_job(Mod, Fun, Tab) ->
    Data = ets:select(Tab, [{{’$1’,’$2’}, [{’/=’, ’$1’, result}], [’$$’]}]),
    Result = Mod:Fun(Data),
    ets:insert({Tab}, {result, Result}).



start(Client) ->
    server:connect(Client),
    ets:new(data, [named_table, public]),
    spawn(?MODULE, input, [self()]),
    loop(data, Client).

loop(Tab, Name) ->
    receive
        {job, {Mod, Fun}} ->
            server:do(Mod, Fun, Tab),
            loop(Tab, Name)
```


Created by XMLmind XSL-FO Converter.

```
    end.

input(Loop) ->
    case read_input() of
        Job ->
            Loop ! {job, Job},
            input(Loop)
    end.

read_input() ->
    [ets:insert(data, Data) || Data <- init_data()],
    returns_the_job_to_be_executed().



start(Client) ->
    server:connect(Client),
    ets:new(data, [named_table, public]),
    spawn(?MODULE, input, [self()]),
    loop(data, Client).

loop(Tab, Name) ->
    receive
        {job, {Mod, Fun}} ->
            server:do(Mod, Fun, Tab),
            loop(Tab, Name)
    end.

input(Loop) ->
    case read_input() of
        Job ->
            Loop ! {job, Job},
            input(Loop)
    end.

read_input() ->
    [ets:insert(data, Data) || Data <- init_data()],
    returns_the_job_to_be_executed().



start(Client) ->
    server:connect(Client),
    ets:new(data, [named_table, public]),
    spawn(?MODULE, input, [self()]),
    loop(data, Client).

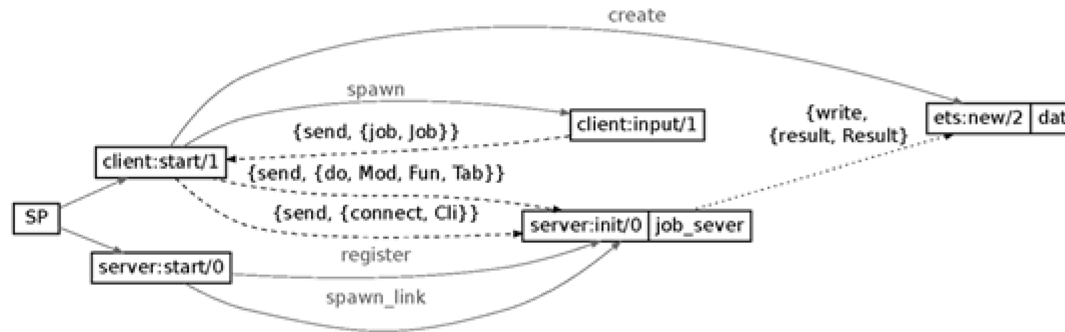
loop(Tab, Name) ->
    receive
        {job, {Mod, Fun}} ->
            server:do(Mod, Fun, Tab),
            loop(Tab, Name)
    end.

input(Loop) ->
    case read_input() of
        Job ->
            Loop ! {job, Job},
            input(Loop)
    end.

read_input() ->
    [ets:insert(data, Data) || Data <- init_data()],
    returns_the_job_to_be_executed().
```

4.10. Hidden Communication

Created by XMLmind XSL-FO Converter.




```
 start(Client) ->
    server:connect(Client),
    ets:new(data, [named_table, public]),
    spawn(?MODULE, input, [self()]),
    loop(data, Client).

loop(Tab, Name) ->
    receive
        {job, {Mod, Fun}} ->
            server:do(Mod, Fun, Tab),
            loop(Tab, Name)
    end.

input(Loop) ->
    case read_input() of
        Job ->
            Loop ! {job, Job},
            input(Loop)
    end.

read_input() ->
    [ets:insert(data, Data) || Data <- init_data()],
    returns_the_job_to_be_executed().



connect(Cli) ->
    ?Name ! {connect, Cli}.

do(Mod, Fun, Tab)->
    ?Name ! {do, Mod, Fun, Tab}.

start() ->
    register(?Name, spawn_link(?MODULE, init, [])).

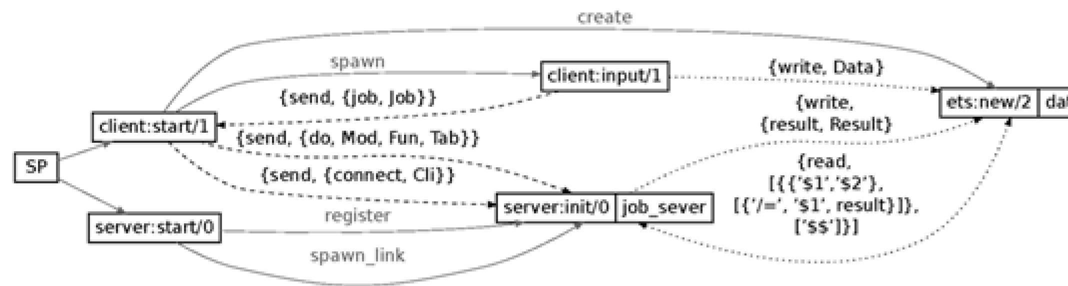
init()->
    ?MODULE:loop([]).

loop(State)->
    receive
        {connect, Cli} ->
            ?MODULE:loop([Cli|State]);
        {do, Mod, Fun, Tab} ->
            handle_job(Mod, Fun, Tab),
            ?MODULE:loop(State)
    end.

handle_job(Mod, Fun, Tab) ->
    Data = ets:select(Tab, [{{'$1','$2'}, [{'/=', '$1', result}], ['$$']}]),
    Result = Mod:Fun(Data),
    ets:insert(Tab, {result, Result}).
```


Created by XMLmind XSL-FO Converter.

4.11. Process Relations



5. Algorithm Description

5.1. Identifying Processes

1.

A process node p_i is created in the graph for each `spawn*` call.

2.

A process node is present in the graph for each function (f) which takes part in communication (when f sends or receives messages or spawns a new process). In this case we have to identify whether the function f already belongs to a process from the first group. Therefore, we calculate the backward call chain of the function f . If the backward call chain contains a spawned function, then the function f belongs to the process of the spawned function p_i . Thus, the communication edges generated by f are linked to p_i .

5.2. Identifying Processes

1.

When a function g takes part in communication, but its backward call chain does not contain a spawned function we create a new process node p_j . This process is identified with the module, the name and the arity of g if there is no communicating function in the backward call chain. Otherwise we select the last communicating function h in the call chain and we identify the created p_j process with module, name and arity of h .

2.

There is a “super process” (SP) in the graph which represent the runtime environment. It represents the fact that the communicating functions can be called from the currently running process, for example from the Erlang shell.

5.3. Creating Process Nodes

1.

At first we collect the spawn expressions from the source code and add them to the set S .

2.

We create a process node p_s for each $s \in S$ spawned process and add it to the set P_s

3.

Created by XMLmind XSL-FO Converter.

We collect the communicating functions C and create process nodes for them (using the second and third step of the identification algorithm).

4.

We link every created p_s ($s \in S$) process to its parent process with a $spawn*$ edge.

5.

We select each register expression from the source code and add the appropriate $register$ link to the graph.

6.

Each process node p_j that has no parent ($p_j \notin P_s$) is linked to the node SP.

5.4. Calculating Communication Edges

1.

We select the message sending expressions from the source code and add it to the set M .

2.

For each $m \in M$ we calculate the receive expression r_m which receives the sent message.

3.

We calculate the containing process node p_m for each $m \in M$ expression and the containing process node p_r for each r_m, and add the $\{send, Message\}$ link from p_m to p_r (where $Message$ is the sent message from the expression m).

5.5. Calculating Hidden Dependencies

1.

The first step is to select the created ETS tables and add them to the set E .

2.

For each $e \in E$ table we create a process node p_e and link it to the parent process. The parent process is the process of the function which calls the function `ets:new/2`.

3.

The next step is to detect whether the found table can be referred using its name. We analyse the option list (the second parameter of the call `ets:new/2`) and calculate its possible values by data-flow reaching. If the `named_table` atom is one of them, then we have to calculate the possible names of ETS table by data-flow reaching, and add the name of the ETS table as an attribute to the process node.

5.6. Calculating Hidden Dependencies

1.

Each ETS table manipulation (e.g. `insert*, delete*`) is added as write operation to the graph between the ETS table node and the process of the expression calling the `ets` functions.

2.

Created by XMLmind XSL-FO Converter.

Each query operation (e.g. `match*, select*`) is added as read operation to the graph between the ETS table node and the process of the expression calling the `ets` functions.

5.7. Calculating write edges

1.

Collect the function calls which refer to an ETS table and change it. Add it to the set W. For example, `ets:insert(Tab, Data)`.

2.

For each $w \in W$ call calculate the referred ETS table with data-flow reaching. At first the possible values of w_1 (denoted with E_w) have to be calculated: $e \in E_w$ and $e \overset{1f}{\rightsquigarrow} w_1$ (where w_1 is the first parameter of the call expression w, e is an expression which value can flow to w_1). If there is an expression $e \in E_w$ which is an atom and its value is $some_name$, then we select the process node (p_e) referring to the named ETS table $some_name$. Otherwise we should find a table reference in E_w which creates the ETS table (a call to `ets:new/2`), and select the process node p_e of the created ETS table.

5.8. Calculating write edges

1.

Determine the process node where the call `ets:insert/2` belongs to: p_w. To identify this process we use a similar algorithm that was presented in the second and third step of the *Identification algorithm*. We determine the function f which contains w, and calculate the process of f.

2.

Connect the process node p_w and the found ETS table node p_e.

5.9. Calculating read edges

1.

Similar to write edge calculation

Created by XMLmind XSL-FO Converter.

Chapter 14. Lecture 14

1. Static Analysis Tools for Erlang

1.1. Erlang Tools

- Support for code understanding and transformations (RefactorErl)
- Type checking (Typer)
- Finding software discrepancies – type errors, unreachable code, unnecessary tests (Dialyzer)
- Refactorings & Duplicated code detection (Wrangler)
- Xref – cross reference analyser

  (http://www.erlang.org/doc/man/xref.html)

2. RefactorErl

2.1. Source code analysis and transformation

- Compile-time syntactic and static semantic analysis of
  - modules, functions, variables, records, etc
  - lifetime, scope, visibility
  - static and dynamic references
  - side effects
  - data-flow, control-flow
- Persistent results, layout, comment preservation
- Acquiring domain-specific knowledge
- Refactoring (> 24)
- Code Comprehension

  www.refactorerl.com

2.2. Source code analysis and transformation

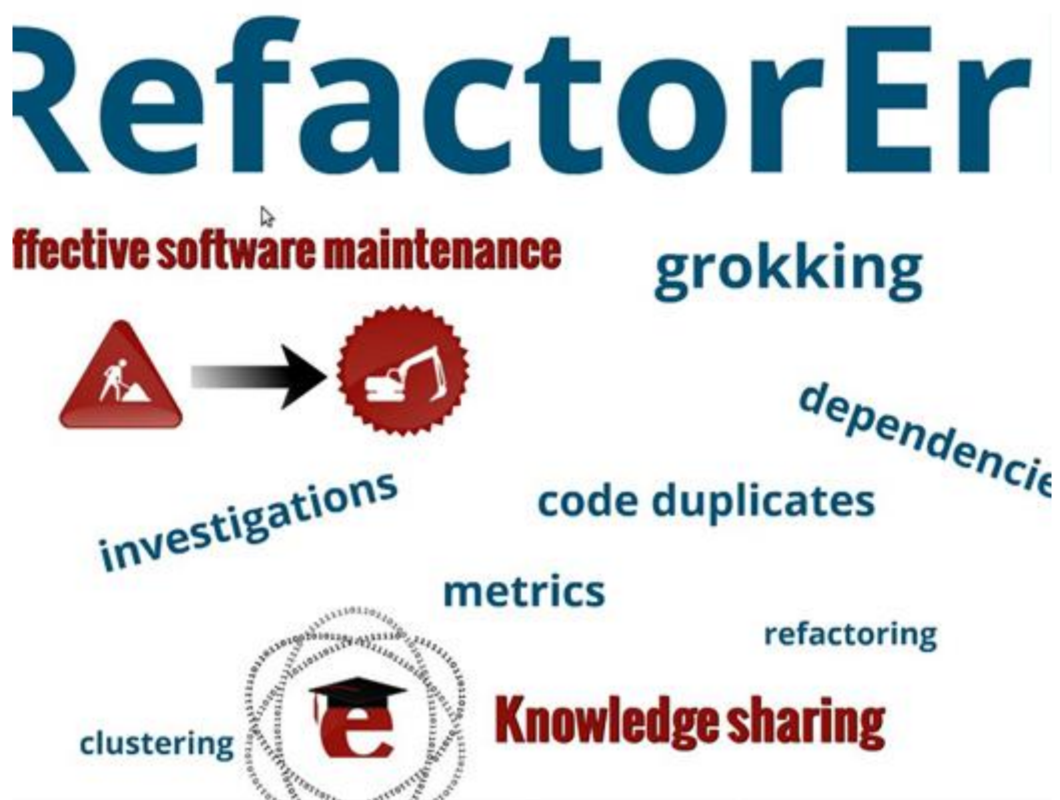
- Compile-time syntactic and static semantic analysis
- Persistent results, layout, comment preservation
- Acquiring domain-specific knowledge
- Refactoring (> 24)
- Code Comprehension:
  - Semantic Queries
  - Software Complexity Metrics

Created by XMLmind XSL-FO Converter.

- Bad smell detection
- Clustering
- Dependency visualistaion

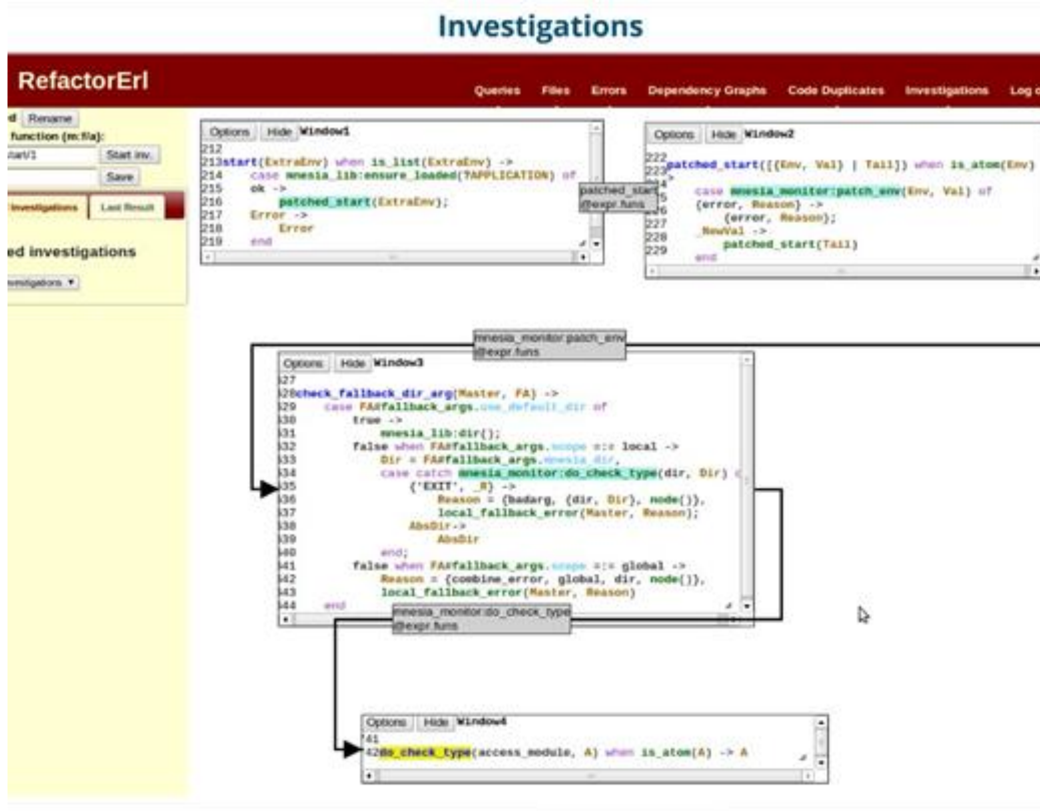
  www.refactorerl.com

2.3. Demo



2.4. Demo

Created by XMLmind XSL-FO Converter.



3. Wrangler

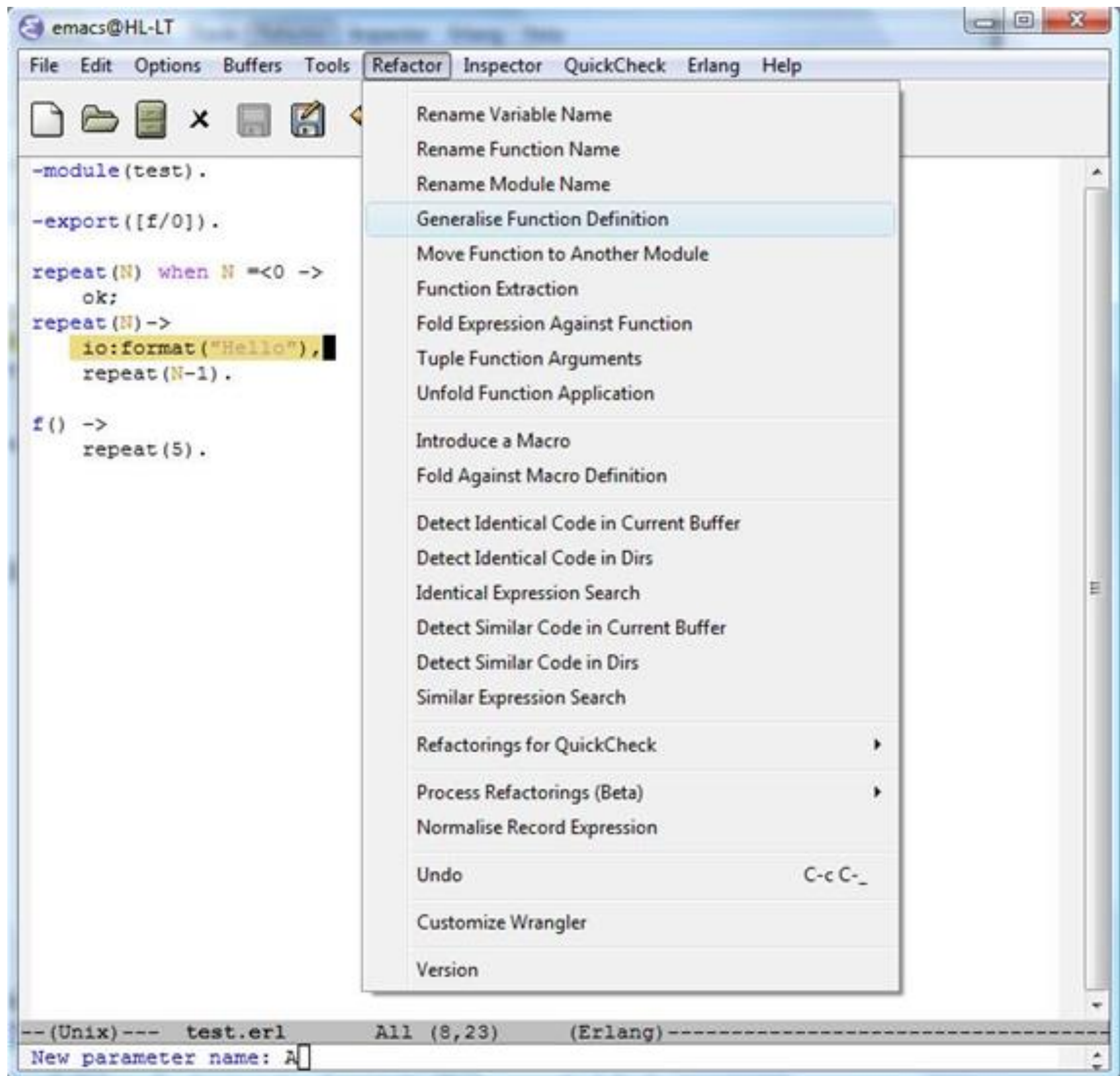
3.1. Wrangler as a Refactoring Tool

- Interactive refactoring tool for Erlang
- Integrated into both emacs and Eclipse.
- Locate and remove code clones
- Extensible:
  - API for writing new refactorings
  - DSL for scripting complex refactoring combinations

    www.cs.kent.ac.uk/projects/wrangler/Home.html

3.2. Demo

Created by XMLmind XSL-FO Converter.



4. Dialyzer & Typer & Tidier

4.1. DIscrepancy AnalYZer for ERlang programs

- Identifies software discrepancies
- Calculates type errors
- Detects code which has become dead or unreachable due to some programming error
- Detects unnecessary tests
- Analyse either debug-compiled BEAM bytecode and Erlang source code
- Uses the “success typing” concept of Typer (no false positives)

  http://www.erlang.org/doc/man/dialyzer.html

  www.user.it.uu.se/ tobiasl/publications/typer.pdf

4.2. The Tidier Refactoring Tool

- Performs simple refactoring steps

Created by XMLmind XSL-FO Converter.

- Web based ui

    http://tidier.softlab.ntua.gr

4.3. Demo

5. Other Tools

5.1. Outside the Erlang World

- Clang

    (http://clang-analyzer.llvm.org/)

- CodeAnalyzer

    (http://www.frontendart.com/product/source-code-analyzer-toolset)

- OpenGrok

    (http://opengrok.github.io/OpenGrok/)

- CodeSurfer

    (http://www.grammatech.com/research/technologies/codesurfer)

5.2. Outside the Erlang World

- Understand

    (http://www.scitools.com/)

- Ariadne

Created by XMLmind XSL-FO Converter.

(http://4dsoft.eu/ariadne)

- CodeCompass
- and so on...

5.3. Why is it important?

"If it was hard to write,

it should be even harder to understand and modify"

Created by XMLmind XSL-FO Converter.

Chapter 15. Practice 1

1. Introduction

1.1. Functional Programming

- The topmost level is a set of modules
- The module is a set of declaration (type, class, function)
- Initial statement
- Evaluation
- Based on mathematical model (Lambda Calculus)
- Turing complete

1.2. Properties

- Referential transparency
- (Static typing)
- Higher-order functions
- (Currying)
- Recursion
- Strict(/lazy) evaluation
- List comprehensions
- Pattern matching
- ("Offset rule")
- IO model

2. Erlang

2.1. Erlang – Properties

- Declarative – Functional programming language, high level of abstraction
- Dynamically typed
- Concurrency – explicit concurrency, LWP
- Soft real-time characteristics
- Robustness – supervison trees
- Distribution – transparent, explicit, network
- Openness, external interfaces – "ports"
- Portability – Unix, Win., ... , heterogeneous network

Created by XMLmind XSL-FO Converter.

- SMP Support – multicore
- “Hot code loading”

2.2. When To Use Erlang?

- Complex, continuously operating, scalable, maintainable, distributed
- Rapid and efficient development
- Fault-tolerant (software, hardware) systems
- Hot-code loading

2.3. Erlang shell

- erl (erl -noshell)
- 1 + 2. "alma".
- q(). – init:stop().
- BREAK menu: Ctrl-C / Ctrl-Break
- User Switch Command: Ctrl-G

2.4. Useful Shell Commands

- help() / h()
- i()
- memory()
- c(ModName)
- ls() / ls(Dir)
- b()
- f() or f(X)
- e(Number) or e(-1)
- v()
- module_name:function_name(Params)
- m(ModName) or module_name:module_info()
- pwd() / cd(Path)

3. Language constructs

3.1. Terms

- Integer – `10, 2#10101, $w`
- Floats – `17.2, 11.12E-10`
- Binaries and Bitstrings – `<<>>, << 0,1,2,3>>, << "hello", 0, "dummy">>`

Created by XMLmind XSL-FO Converter.

- Atom – `atom1, is_atom@, 'Atom 1'`
- Boolean – true, false
- Tuple – `{}, {1,2}, {a, 2}`
- List – `[], [1, {foo, 2}, [bar, nok]]`
- String – `"A string"`, `[97,98,99,100] == "hello" == [$a, $b, $c, $d]`
- Unique identifiers: pid (`< 0.4.2>`), port (`#Port<0.472>`), reference (`#Ref<0.0.0.42>`)
- Fun – `#Fun<...>`

3.2. Comparison Of Types

number < atom < fun < port < pid < tuple < list < binary

- `<, >`
- `=<, >=`
- `/=, ==`
- `=:=, =/=`

3.3. Arithmetic, Bit and Logical operators

- `+, -, *, /, div, rem`
- `and, or, not`
- `andalso, orelse`
- `bsl, bsr, band, bor, bxor, bnot`

3.4. Variables and Pattern Matching

- Start with an Upper Case Letter or underscore character
- Single assignment without declaration
- Variables are local to a function clause
- Pattern Matching:
  - Controlling the execution flow
  - Assigning values to variables
  - Extracting values from compound data structures
  - $X = 2$
  - $\{A, A, X\} = \{1, 1, 3\}$
  - $[Head|Tail] = [1, 2, 3]$

3.5. Modules

- Name of the module is an atom

Created by XMLmind XSL-FO Converter.

- File extension: “.erl”
- Sequence of forms (attributes and function, record and macro definitions)
- Compile: c(mod), erlc mod.erl

3.6. Attributes

- `-module(alma)`
- `-export([f/1]), -import(lists, [max/1])`
- `-include("a.hrl”)`
- `-compile(export_all)`

3.7. Functions – ModName:FunName/Arity

```
  fact(0) -> 1;
fact(N) when N>0 ->
    N * fact(N-1).

fib(1) -> 1;
fib(2) -> 1;
fib(N) when N>2 ->
    A = fib(N-1),
    B = fib(N-2),
    A + B.
```

3.8. Built In Functions (BIF)

- Implemented in C
- Interface in module “erlang”
- Low level operations, performance
- Functions with side-effect
- etc.
- hd/1, tl/1, length/1, size/1, element/2, setelement/2, list_to_tuple/1, ...

3.9. Lists

- ’- -’ and ’++’
- length/1, tl/1, hd/1
- lists: {max/1, sum/1, nth/2, last/1, reverse/1, member/2, delete/2, sort/1, usort/1, zip/2, split/2}

```
  quicksort([])   -> [];

quicksort(List) ->
  quicksort([X || X <- List, X =< Y])
    ++ [Y]
      ++ quicksort([X || X <- List, X > Y])
```

3.10. Conditional Evaluation – case, if

Created by XMLmind XSL-FO Converter.

```
    case f(X) of
      [H|T] when H > 0 -> H;
      []               -> 0;
      {A,B} when A > B -> A;
      _                ->
          X = nok,
          X
  end

  if
      A > B -> A;
      A < B -> B;
      true  ->
          X = eq,
          X
  end
```

3.11. Guard Expressions

- ',' and ';'
- Bound variables
- Literal
- Comparison
- Arithmetic expression
- Boolean expression
- Type checking
- guard built in functions (subset of BIFs)

3.12. Fun Expressions

```
    fun({A,B}) when A > B ->
          Z = A*B,
          {A, B, Z};
      ({A}) ->
          {A, A, A};
      ({}) ->
          {0,0,0}
  end

  fun mymod:myfun/5
```

3.13. Dynamic constructs

```
  apply(ModName, FunName, [Param1, ..., ParamN])
Mod:Fun(Param1, ..., ParamN)

apply(FunExpr, [Param1, ..., ParamN])
Fun(Param1, ..., ParamN)
```

3.14. Trapping Run-time Errors

```
  try
```


Created by XMLmind XSL-FO Converter.

```
        {F, A} = lists:keyfind(F, 1, M:module_info(exports)),
        A
    of
        X ->
            Param = read_params(X, X),
            apply(M, F, Param)
    catch
        error:undef ->
            io:format("Module does not exists: ~p~n", [M]);
        error:{badmatch, _} ->
            io:format("Function ~p does not present in modulban ~p~n", [F, M]);
        Class:Type ->
            io:format("Unhandled error occured: ~p:~p~n", [Class, Type])
    after
        io:format("After branch~n", [])
    end.
```

3.15. Records

```
     -record(person, {name = "",
                      date={1900, 1, 1},
                      address}).

    Rec    = #person{name="Mr. Smith", address = "Budapest"}
    NewRec = Rec#person{date={1950,10,10}}

    NewRes#person.name
    #person.name

    case Rec of
        #person{name = "Mr. Smith", date = D} ->
            D;
        Rec2 = #person{}->
            Rec2#person.date;
        _ ->
            bad_record
    end
```

3.16. Macros

```
     -define(mymac, 42).
   -define(mymac(Par1, Par2), Par1+Par2)

   f() ->
       ?mymac(42, ?mymac).


   -undef(mymac).

   -ifndef(mymac). %% -ifdef(mymac).
       ...
   -else.
       ...
   -endif.
```


Created by XMLmind XSL-FO Converter.

Chapter 16. Practice 2

1. Concurrent Erlang

1.1. Processes

- Actor with separated memory space (own heap and stack)
- Do not share memory
- Own state
- Communication with asynchronous message passing

1.2. Example Ping-Pong Server

```
  run() ->
  Pid = spawn(fun ping/0),
  Pid ! self(),
  receive
    pong -> ok
  end.

ping() ->
  receive
    From -> From ! pong
  end.
```

1.3. Concurrent language elements

Spawning processes:

```
    Pid = spawn(ModName, FunName, [Args])
```

Sending messages:

```
    Pid ! {some_message, Msg} %% send/2
```

Receiving messages:

```
    receive
      {A, B, C} ->           A;
      {some_message, Msg} -> Msg
  after
      1000 ->                nok
  end
```

General:

```
  self()
pid(0,42,0)
```

1.4. Process links and error handling

Created by XMLmind XSL-FO Converter.

- `link/1, spawn_link/3`
- exit signal, if process terminates– normal or non-normal
- `process flag(trap_exit, true)`
- `{'EXIT', Pid, Reason}` message
- `unlink/1`
- `exit(Reason), exit(Pid, Reason)` – normal, kill, other
- supervision

1.5. Registering processes

```
  register(foo, Pid)
foo ! {msg, "Final message''}

unregister(foo)
whereis(foo)
```

1.6. Erlang Term Storage – ETS(1)

- %% Shared memory
- Key-Value storage for large quantities of data
- Constant time access
- Not a KV Database, no transactions
- `TableId = ets:new(TableName, [Options])`
- Options: named_table, set, bag, ordered_set, duplicate_bag, private, protected, public, keypos, Key, (read)write_concurrency
- `ets:delete(TableId)`
- `ets:insert(TableId, Key, Value)`
- `ets:lookup(TableId, Key)`
- `ets:delete(TableId, Key)`

1.7. Erlang Term Storage – ETS(2)

- `ets:first(TableId), ets:next(TableId), '$end_of_table'`
- `ets:match(TableId, Pattern) - $1, $0`
- `ets:match_object(TableId, Pattern)`
- `ets:delete_object(TableId, Pattern)`
- `ets:select(TableId, MatchSpec)`

2. Distributed Erlang

2.1. Distributed Erlang Nodes

Created by XMLmind XSL-FO Converter.

- Starting the node:

  `erl -name foo@host`

  `erl -sname foo -setcookie bar`

- `node(), nodes()`
- Message passing:

  `{Name, Node} ! {msg, "Final message”}`

- Connection:

  `net_adm:ping(Node),`

  `monitor_node(Node, true)`

- ps ax | grep -i epmd
- "Magic Cookie":

  `get_cookie(), set_cookie(node(), "bar")`

- Remote shell: Ctrl-G, r, c

3. Advanced topics

3.1. Ports and Port Drivers

- Connection with the ”world outside“
- “World outside”: an OS process
- Receiving and sending messages:standard input/output
- Byte-oriented communication
- `open_port(PortName, PortSettings) - {spawn, Command}, {packet, N}`
- `{Pid, command, Data}` – send data to port
- `{Pid, close}, {Pid, {connect, NewPid}}`
- `{Port, {data, Data}}` – receive data from port
- `{Port, closed}, {Port, connected}, {’EXIT’, Port, Reason}`
- BIF: `port_command/2, port_close/1, port_connect/2, ports/0, port_info/2`

3.2. Connection with other languages

- ErlInterface
- jinterface (Jungerl)
- OTP.NET

3.3. Nice features

- Software Upgrades
- OTP behaviours

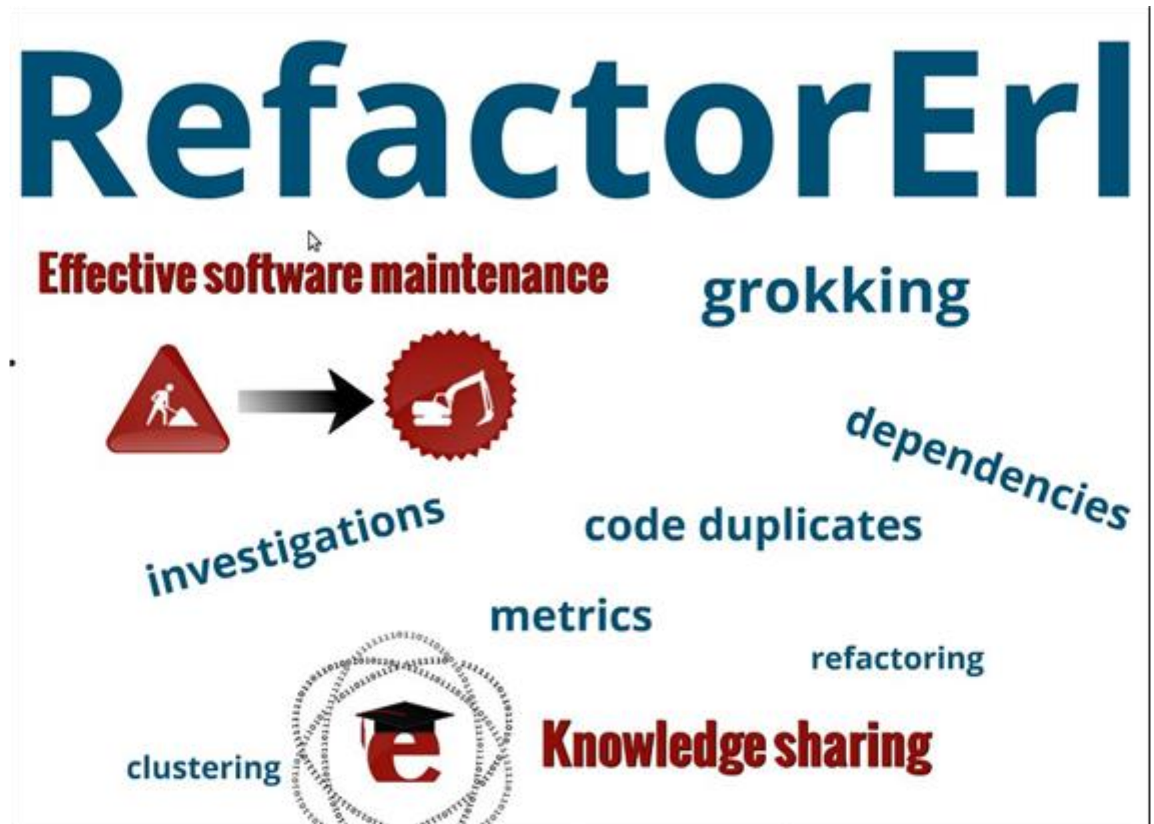
Created by XMLmind XSL-FO Converter.

- Supervision Trees
- NIF
- Web Servers
- DBMS

Created by XMLmind XSL-FO Converter.

Chapter 17. Practice 3

1. RefactorErl



1.1. History

- Started in 2006
- Ericsson Software Technology Lab (2011)
- University and ELTE-Soft Staff
- PhD, MSc, BSc students
- Supported by
  - Ericsson Technology Hungary
  - National Technology Program TECH_08_A2-SZOMIN08
  - KMOP-1.1.2-08/1-2008-0002
  - GVOP-3.2.2-2004-07-0005/3.0 ELTE IKKK
  - EITKIC 12-1-2012-0001
  - Paraphrase Enlarged (Seventh Framework Program)

1.2. Motivation

- Size of an industrial application grows fast

Created by XMLmind XSL-FO Converter.

  - Split applications into smaller parts
  - Split source files ...
- Simple tasks become hard to perform
  - Rename a function/variable/macro...
  - Generalise a function
  - Eliminate duplicated code parts
- Source code does not fit into coding conventions, does it?
- New developers arrive to the project
  - What does the program do?
  - What is the relation among different source code parts?
  - Where is a given function defined?

1.3. Our solution

- Research of Erlang refactoring
- Static source code analyser and transformer tool
  - Refactoring – less error prone & fast
  - Program comprehension
- Real-world applications for analysis
- Support in everyday work & debugging & complex tasks, e.g.
  - Rename a function, search definition
  - Find the value of a variable
  - Program comprehension, component relation detection
  - Program restructuring

1.4. Requirements

- Work with large code base
- Language coverage
- Comment preservation
- Layout preservation (indentation)

1.5. Design goals

Large source code base have to be analysed!

1.

Store semantic information instead of calculating each time

- Efficient retrieval – graph model

Created by XMLmind XSL-FO Converter.

- Incremental analysis

2.

Provide a platform for source code transformation

- Generic solutions are preferred
- Non-refactoring applications

1.6. Emerged research topic

- Efficient graph model to store semantic information
- Behaviour preservation during transformation
- Validate refactorings
- Static source code analysis: call graph (including dynamic calls), data-flow analysis, side-effect analysis, message passing analysis, analysing preprocessor constructs, etc
- Make the result of the analysis available for the developers

2. Architecture

2.1. Three-layered graph model

1.

Lexical level

- tokens
- preprocessing
- comments, whitespace

2.

Syntax level

- abstract syntax tree

3.

Semantic level

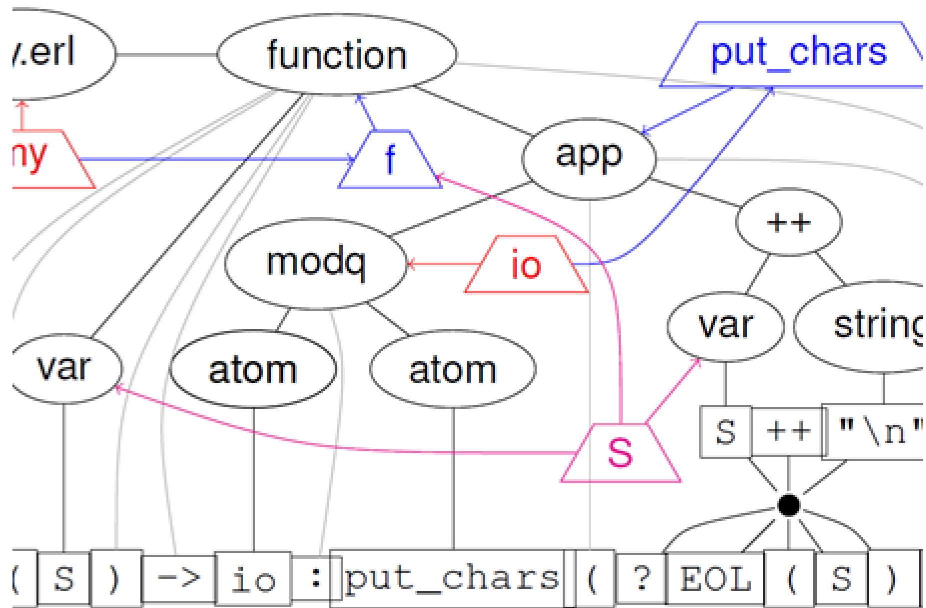
- module, function, record, variable usage and references
- side-effect, dynamic call information, data-flow, etc.

```
-module(my).

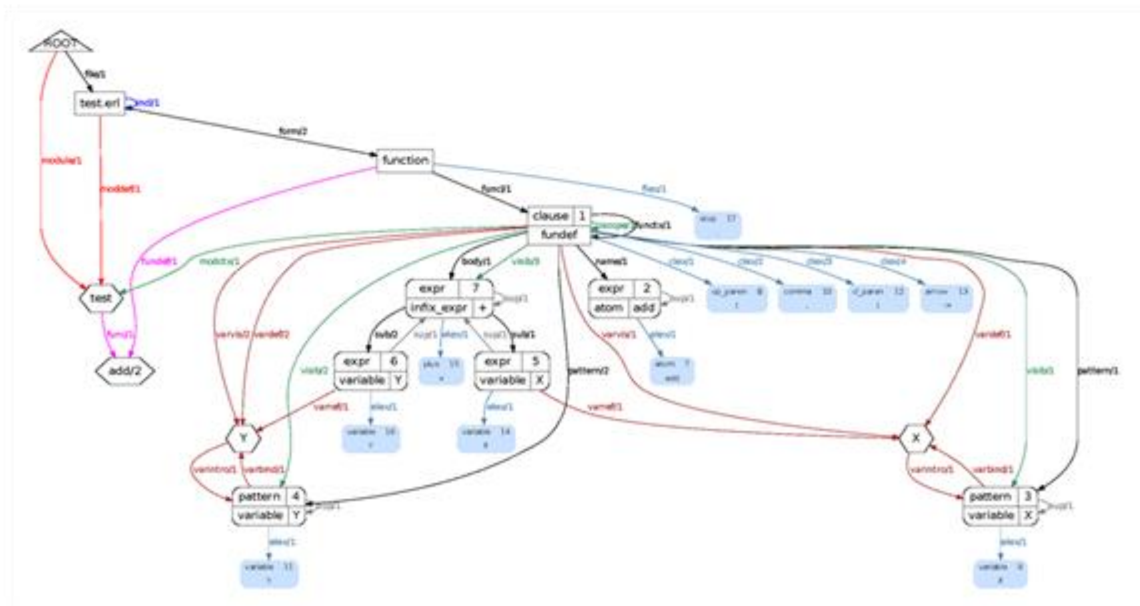
-define(EOL(X), X ++ "\n").

f(S) -> io:put_chars(?EOL(S)).
```


Created by XMLmind XSL-FO Converter.



2.2. Example graph for `add/2`



2.3. Graph storage

- Nodes and edges are stored in databases:
  - Mnesia, C++ graph, Kyoto Cabinet
  - Node attributes: token text, variable name, …
  - Edge labels: subexpression, variable reference, …
- Graph path: filtered edge label sequence
  - Edges are indexed by label
  - Cost doesn't grow with code size
- Frequently used queries need only fixed length paths

Created by XMLmind XSL-FO Converter.

2.4. Other details

- Extended syntax description
  - Defines the representation
  - Source for parser, lexer, and token updater
- Analyser framework
  - Extensible, modular structure
  - Works on syntactic subtrees (incremental)
  - Asynchronous parallel execution
  - Side-effect analysis, data-flow analysis, dynamic function call analysis

3. Features

3.1. Features

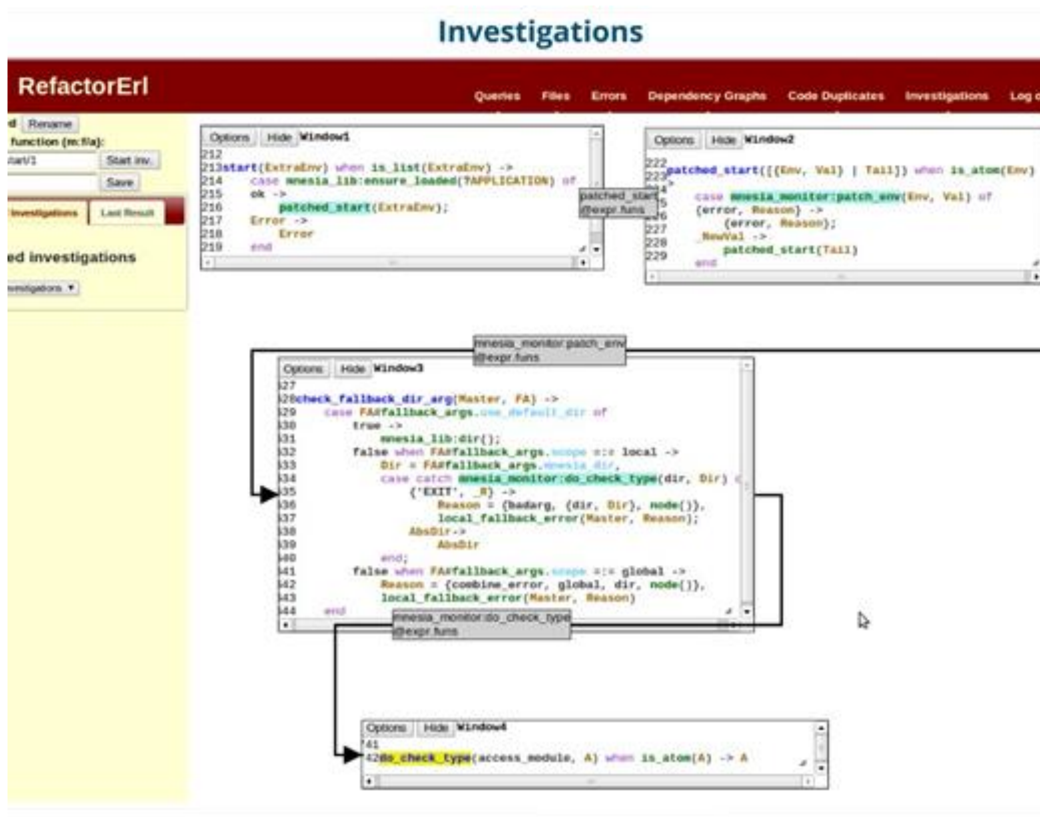
- Compile-time syntactic and static semantic analysis of
  - modules, functions, variables, records, etc
  - lifetime, scope, visibility
  - static and dynamic references
  - side effects
  - data-flow, control-flow
- Refactoring (> 24)
- Code Comprehension:
  - Semantic Queries
  - Software Complexity Metrics
  - Bad smell detection
  - Clustering
  - Dependency visualisation

3.2. User Interfaces

- Emacs, XEmacs, Vi, Eclipse plugins
- Command line interfaces:
  - Scriptable and interactice Erlang shell interface
  - CLI
- Desktop GUI (based on WXErlang)
- Web based interfaces (based on Nitrogen/Angular and YAWS)

Created by XMLmind XSL-FO Converter.

3.3. Web UI



3.4. Where to find us?

http://refactorerl.com

http://pnyf.inf.elte.hu/trac/refactorerl/

4. Install & Configure

4.1. Installation and configuration

- What you need: Erlang (R13B – R16B)
- Optional requirements:
  - C++ compiler (g++ 4.3.6 or a newer version)
  - Yaws 1.89 – 1.96
  - Graphviz
  - Editors: Emacs, Vim, Eclipse

4.2. Installation and configuration

- `make` or `bin/referl -build tool`
- `bin/referl -build tool -no_cpp`
- `bin/referl -build tool -yaws_189`

Created by XMLmind XSL-FO Converter.

4.3. Starting the tool

- `bin/referl`
- `bin/referl [Options]`
- `bin/referl -help`

4.4. Start Options

`-build TARGET`	Build TARGET (e.g. tool, doc, clean)
`-bufsrv`	Build bufferserver (use with '-build tool')
`-client`	Start in client mode (no server is started)
`-db DB`	[mnesia|nif|kcmini] The database engine to use (default: mnesia)
`-dir DIR`	Sets the RefactorErl data directory
`-base PATH`	Path to the RefactorErl base directory
`-pos POS`	[abs|rel] The positioning mode to use (default: abs)
`-erl PATH`	Path to the Erlang executable to use
`-g++ PATH`	Path of the g++ compiler to use
`-synchronize`	Database synchronization
`-help`	Print this help text

4.5. Start Options

`-server`	Start in server mode (no shell is started)
`-sname NAME`	Short name of the Erlang node
`-name NAME`	Full name of the Erlang node
`-srvname NAME`	Name of the Erlang server node to connect
`-client`	Start in client mode (no server is started)
`-emacs`	Start as an Emacs client
`-vim`	Start as a Vim client
`-wx`	Start as a Wx client

4.6. Start Options

`-nitrogen`	Start with Nitrogen
`-web2`	Start with Web2
`-yaws_path PATH`	Path to the Yaws ebin directory

Created by XMLmind XSL-FO Converter.

	(need /ebin at the end)
`-yaws_name NAME`	Set Yaws server name
`-yaws_port PORT`	Set Yaws port
`-yaws_listen IP`	Set Yaws IP
`-browser_root`	Set the file browser root directory
`-images_dir`	Set root directory where generated Nitrogen images will be written
`-restricted_mode`	Set restricted mode on the web interface or on a RefactorErl started as server

Created by XMLmind XSL-FO Converter.

Chapter 18. Practice 4

1. Analysing Erlang Modules

1.1. Building the Database

- `ri:add("path_to_file_or_dir”)`

  – recursive

- `ri:add(“test.erl”)`
- Adding applications
  - `ri:envs()`
  - `ri:addenv(appbase, path_to_app)`
  - `ri:add(usr, mnesia)`
- Include files
  - `ri:addenv(include, path_to_includes)`

    – not recursive

2. The Semantic Program Graph

2.1. Three-layered graph model

1.

   Lexical level

   - tokens
   - preprocessing
   - comments, whitespace

2.

   Syntax level

   - abstract syntax tree

3.

   Semantic level

   - module, function, record, variable usage and references
   - side-effect, dynamic call information, data-flow, etc.

2.2. Lexical Schema

```
-define(LEXICAL_SCHEMA,
       [{lex,    record_info(fields, lex),
                 [{mref, form},
```


Created by XMLmind XSL-FO Converter.

```
                    {orig, lex}, {llex, lex}]},
          {file,   [{incl, file}]},
          {form,   [{iref, file}, {flex, lex},
                    {forig, form}, {fdep, form}]},
          {clause, [{clex, lex}]},
          {expr,   [{elex, lex}]},
          {typexp, [{tlex, lex}]}
         ]).

-record(lex,      {type, data}).
-record(token,    {type, text, prews="",
                   postws="", scalar, linecol}).
```

2.3. Syntactic Schema

```
-define(SYNTAX_SCHEMA,
        [{root,   [],
                  [{file, file}]},
         {file,   record_info(fields, file),
                  [{form, form}]},
         {clause, record_info(fields, clause),
                  [{body, expr}, {guard, expr},
                   {name, expr},
                   {pattern, expr}, {tmout, expr}]},
         {expr,   record_info(fields, expr),
                  [{aftercl, clause}, {catchcl, clause},
                   {esub, expr},
                   {exprcl, clause}, {headcl, clause}]},
         {form,   record_info(fields, form),
                  [{eattr, expr}, {funcl, clause},
                   {tattr, typexp}]},
         {typexp, record_info(fields, typexp),
                  [{texpr, expr}, {tsub, typexp}]}]).
```

2.4. Syntactic Schema

```
 -record(file,     {type, path, eol, lastmod, hash}).
-record(form,      {type, tag, paren=default,
                    pp=none, hash, form_length,
                    start_scalar, start_line}).
-record(clause,    {type, var, pp=none}).
-record(expr,      {type, role, value, pp=none}).
-record(typexp,    {type, tag}).
```

2.5. Semantic Schema

```
 refanal_mod:schema/0
    [{module, record_info(fields, module),
              []},
     {root,   [{module, module}]},
     {file,   [{moddef, module}]},
     {clause, [{modctx, module}]}
    ]
refanal_fun:schema/0
    [func,    record_info(fields, func),
              [{funcall, func}, {dyncall, func},
               {ambcall, func}, {may_be, func}],
     {form,   [{fundef, func}]},
     {clause, [{functx, clause}]},
     {expr,   [{modref, module}, {funeref, func}, {funlref, func},
               {dynfuneref, func}, {ambfuneref, func},
               {dynfunlref, func}, {ambfunlref, func},
```


Created by XMLmind XSL-FO Converter.

```
                   {localfundef, func}]},
         {module, [{func, func}, {funexp, func}, {funimp, func}]}
        ]
    refanal_ets:schema/0
        [{ets_tab,record_info(fields, ets_tab),
                  [{ets_ref, expr}, {ets_def, expr}]}]
```

2.6. Semantic Schema

```
    refanal_var:schema/0
        [{variable, record_info(fields, variable),
                    [{varintro, expr}]},
         {clause,   [{scope, clause}, {visib, expr},
                     {vardef, variable}, {varvis, variable}]},
         {expr,     [{varref, variable}, {varbind, variable}]}
        ]
    refanal_expr:schema/0
        [{expr, [{top, expr}, {clause, clause}]}]
    refanal_rec:schema/0
        [{field,  record_info(fields, field),
                  []},
         {typexp, [{fielddef, field}]},
         {expr,   [{fieldref, field}]},
         {record, record_info(fields, record),
                  [{field, field}]},
         {file,   [{record, record}]},
         {form,   [{recdef, record}]},
         {expr,   [{recref, record}]}
        ].
```

2.7. Semantic Schema

```
     -record(module,   {name}).
    -record(record,   {name}).
    -record(field,    {name}).
    -record(func,     {name              :: atom(),
                       arity             :: integer(),
                       dirty  = int      :: no | int | ext,
                       type   = regular  :: regular | anonymous,
                       opaque = false    :: false | module |
                                            name | arity}).
    -record(variable, {name}).
    -record(env,      {name, value}).
    -record(ets_tab,  {names}).
    -record(pid,      {reg_name}).
```

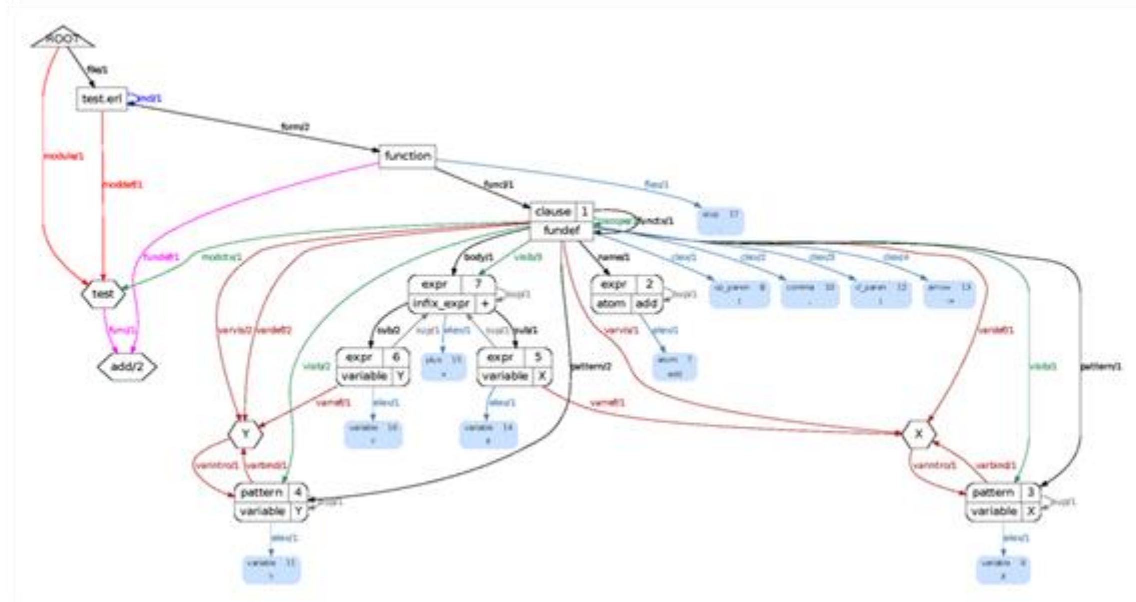
2.8. Simple Erlang File

test.erl

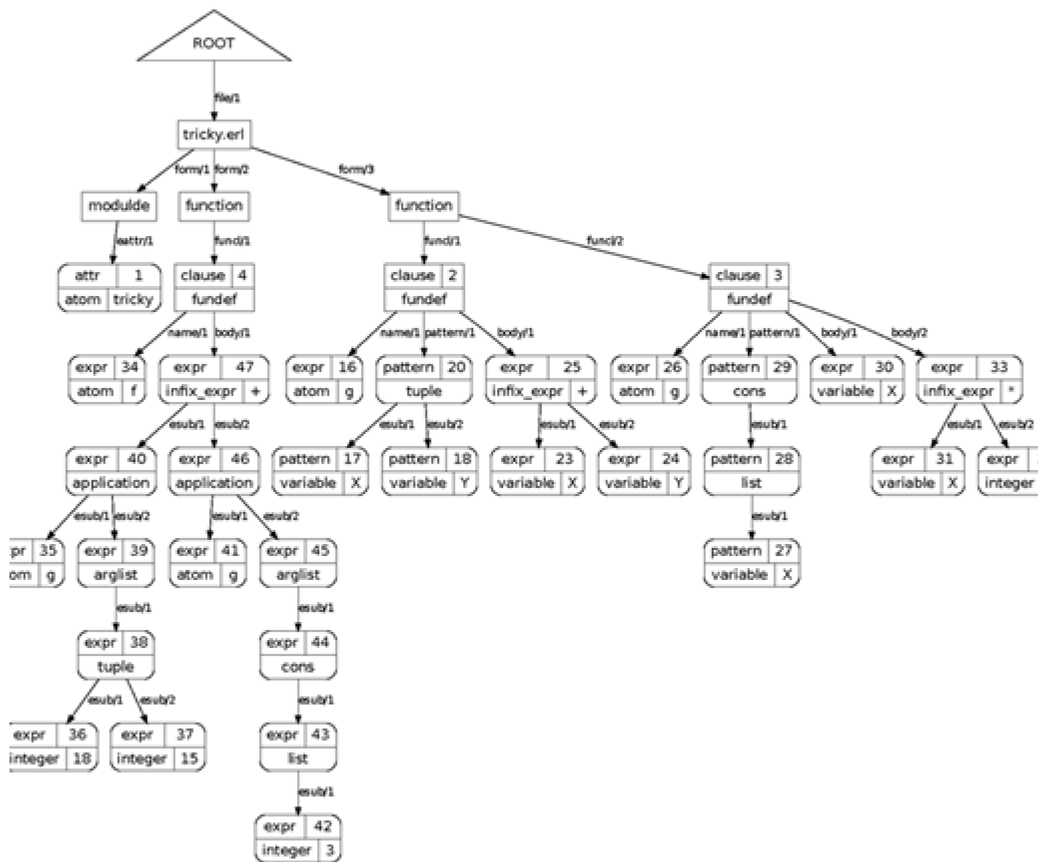
```
     -module(test).

   add(X,Y) ->
       X+Y.
```

2.9. Example graph for `add/2`



2.10. Reproduce the Original Source File



3. Graph Traversal

3.1. Path expressions

- Supports information gathering for refactoring and high level analysis

Created by XMLmind XSL-FO Converter.

- Depends on the representation

```
path() = [PathElem]

 PathElem = Tag | {Tag, Index} | {Tag, Filter} |
            {intersect, node(), Tag}
 Tag      = atom() | {atom(), back}
 Index    = integer() | {integer(), integer()} | {integer(), last}
 Filter   = {Filter, 'and', Filter} | {Filter, 'or', Filter} |
            {'not', Filter} | {Attrib, Op, term()}
 Attrib   = atom()
 Op       = '==' | '/=' | '=<' | '>=' | '<' | '>'

refcore_graph:path(StartNode, [PathElem])
```

3.2. Path expression example

- List the analysed files:

```
    path(Root, [file])

Root = refcore_graph:root()
```

- List the defined functions from file `Mod`:

```
    path(Mod, [{form, {type, '==', func}}])
    %% syntactic
path(Mod, [moddef, func])
    %% semantic
```

3.3. Exercises

- Select every function definition form:
-

```
path(Root, [file, {form, {type, '==', func}}])
```

- List the top level case expressions:
-

```
path(Root, [file,
        {form, {type, '==', func}},
        funcl,
        {body, {type, '==', case_expr}}])
```

3.4. Exercises

- Find every atom `apple`:

Created by XMLmind XSL-FO Converter.

-

```
path(Root, [file,
        {form, {type, '==', func}},
        funcl,
        visib,
        {{top, back},
         {value, '==', apple}}])
```

- This is just a partial solution, solve it with a recursive Erlang function!

3.5. Exercises

- Find every string expression containing the “Error” substring!
- Find the definition and the references of the function `mnesia_log:open_log/4`!
- Find the function that calls the function `mnesia_log:open_log/4` outside from its defining module!
- Find the record definitions and usages!
- Find the variable binding points (expressions) for the variable `Log`!

3.6. Query Library

- Set of library module: `reflib_clause`, `reflib_file`, `reflib_expression`, `reflib_form`, `reflib_macro`, `reflib_dynfun`, `reflib_function`, `reflib_module`, `reflib_record`, `reflib_record_field`, `reflib_token_gen`, `reflib_variable`
- Extended evaluation framework: `reflib_query`:

```
    query(Start, End) =
    refcore_graph:path() |
    fun(node(Start)) -> [node(End)] |
    tuple()
exec/1, exec/2, exec1/2, exec1/3
seq/1, seq/2, all/1, all/2,
any/1, any/2, unique/1
```

- Example: `exec(Mod, reflib_module:function())`

3.7. Query example

- List the analysed files:

```
?Qurey:exec(reflib_file:all())
```

- List the defined functions from file `Mod`:

```
?Query:exec(Mod,
        ?Query:seq([reflib_file:module(),
                    reflib_module:locals()]))
```

3.8. Exercises

Created by XMLmind XSL-FO Converter.

- Select every function definition form:
-

```
?Query:exec(
?Query:seq([reflib_module:all(),
            reflib_module:locals(),
            reflib_function:definition()]))
```

3.9. Exercises

- List the top level case expressions:
-

```
    A =
?Query:exec(
    ?Query:seq([reflib_file:all(),
                reflib_file:forms(),
                reflib_form:clauses(),
                reflib_clause:body()]))
lists:filter(
    fun(E) ->
        reflib_expression:type(E) == case_expr
    end, A).
```

3.10. Exercises

- Find every atom `apple`:
-

```
    Exprs = ?Query:exec(
            ?Query:seq([?Mod:all(),
                        ?Mod:locals(),
                        ?Fun:definition(),
                        ?Form:clauses(),
                        ?Clause:body(),
                        ?Expr:deep_sub()])),
lists:filter(fun(E) ->
                 ?Expr:type(E) == atom
                   andalso
                 ?Expr:value(E) == apple
             end, Exprs)
```

- See the implementation of `deep_sub`!

3.11. Exercises

- Find every string expression containing the “Error” substring!
- Find the definition and the references of the function `mnesia_log:open_log/4`!
- Find the function that calls the function `mnesia_log:open_log/4` outside from its defining module!
- Find the record definitions and usages!

Created by XMLmind XSL-FO Converter.

- Find the variable binding points (expressions) for the variable `Log`!

Created by XMLmind XSL-FO Converter.

Chapter 19. Practice 5

1. Language Definition

1.1. Semantic query language

- A user level query language to get information about the Erlang source
- Language concepts:
  - Entities
  - Selectors
  - Properties
  - Filters
- Initial selection + query sequence: selection, iteration, closure and property query
- Metrics built in the query language as property
- Custom query or predefined query
- Example:

  `mods[name==mymod].funs[name==myfun].calls`

  `@file.funs[name==myfun].calls`

1.2. Syntax of the queries

```
 <semantic query> ::=
    <initial selector> ('['<filter>']')* ('.' <subquery>)*
    ['.' <property> [':' <statistics>]]

<subquery> ::=
    <selector> ('['<filter>']')* |
    <iteration> ('['<filter>']')* |
    <closure> ('['<filter>']')*

<iterĂĄciĂł> ::=
    '{' <subquery> ('.' <subquery>)* '}' int

<closure> ::=
    '(' <subquery> ('.' <subquery>)* ')' int |
    '(' <subquery> ('.' <subquery>)* ')' '+'
```

1.3. Syntax of the queries

- Filters are logical expressions or
- embedded queries (checking whether the result is empty or not)

Precedence of the operators (decreasing)	
Operator	*Associativity*

Created by XMLmind XSL-FO Converter.

`not`	
`/= == = >= =< < > =:= =/= like in`	left
`and`	left
`or`	left

2. Language Elements

2.1. Entities

- file (module)
- function
- function clause
- variable
- record
- record filed
- expression
- macro

2.2. Initial Selectors

Name	*Synonym*	*Type*
`@function`	`@fun`	function
`@clause`	-	clause
`@variable`	`@var`	variable
`@macro`	-	macro
`@record`	`@rec`	record
`@recfield`	`@field`	record field
`@file`	-	file
`files`	-	file
`@module`	`@mod`	file
`modules`	`mods`	file
`@expression`	`@expr`	expression

Created by XMLmind XSL-FO Converter.

`@definition`	`@def`	any

2.3. File Selectors

Name	*Synonyms*	*Type*
`functions`	`function, funs, fun`	function
`records`	`record, recs, rec`	record
`macros`	`macro`	macro
`includes`	-	file
`included_by`	-	file
`imports`	-	function
`exports`	-	function

2.4. File Properties

Name	*Synonyms*	*Type*
`is_module`	`is_mod, module, mod`	bool
`is_header`	`header`	bool
`name`	-	atom
`filename`	-	string
`directory`	`dir`	string
`path`	-	string

2.5. Function Selectors

Name	*Synonyms*	*Type*
`references`	`reference, refs, ref`	expression
`dynamic_references`	`dynrefs, dynref`	expression
`calls`	-	function

Created by XMLmind XSL-FO Converter.

dynamic_calls	dyncalls, dyncall	function
called_by	-	function
dynamic_called_by	dyncalled_by	function
arguments	argument, args, arg, parameters, parameter, params, param	expression
expressions	expression, exprs, expr	expression
variables	variable, vars, var	variable
file	-	file

2.6. Function Properties

Name	*Synonyms*	*Type*
exported	-	bool
name	-	atom
arity	-	int
bif	-	bool
pure	-	bool
defined	-	bool
is_dirty	dirty	bool
module	mod	atom

2.7. Function Clause Selectors

Name	*Synonyms*	*Type*
calls	-	function
dynamic_calls	dyncall, dyncalls	function
arguments	argument, args, arg, parameters, parameter, params, param	expression
body	-	expression

Created by XMLmind XSL-FO Converter.

expressions	expression, exprs, expr	expression
variables	variable, vars, var	variable
file	-	file
function	func	function

2.8. Function Clause Properties

Name	*Synonyms*	*Type*
name	-	atom
arity	-	int
module	mod	atom
index	-	int

2.9. Expression Selectors

Name	*Synonyms*	*Type*
fundef	-	function
functions	function, funs, fun	function
dynamic_functions	dynamic_function, dynfuns, dynfun	function
variables	variable, vars, var	variable
records	record, recs, rec	record
macros	macro	macro
subexpression	subexpr, esub, sub	expression
parameter	param	expression
top_expression	top_expr, top	expression
reach	-	expression
origin	-	expression
file	-	file

Created by XMLmind XSL-FO Converter.

2.10. Expression Properties

Name	*Synonyms*	*Type*
`type`	-	atom
`value`	`val`	any
`class`	-	atom
`is_last`	`last`	bool
`has_side_effect`	-	bool
`is_tailcall`	`tailcall`	bool
`tuple_repr_of_record`	`record_tuple`	bool

2.11. Variable Selectors

Name	*Synonyms*	*Type*
`references`	`reference, refs, ref`	expression
`bindings`	-	expression
`function_definition`	`fundef`	function

2.12. Variable Properties

Name	*Synonyms*	*Type*
`name`	-	string

2.13. Record Selectors

Name	*Synonyms*	*Type*
`references`	`reference, refs, ref`	expression
`fields`	`field`	record field
`file`	-	file

Created by XMLmind XSL-FO Converter.

2.14. Record Properties

Name	*Synonyms*	*Type*
`name`	-	atom

2.15. Record Field Selectors

Name	*Synonyms*	*Type*
`references`	`reference, refs, ref`	expression
`record`	`rec`	record
`file`	-	file

2.16. Record Field Properties

Name	*Synonyms*	*Type*
`name`	-	atom

2.17. Macro Selectors

Name	*Synonyms*	*Type*
`references`	`reference, refs, ref`	expression
`file`	-	file

2.18. Macro Properties

Name	*Synonyms*	*Type*
`name`	-	string
`aritiy`	-	int
`const`	-	bool

Created by XMLmind XSL-FO Converter.

2.19. Statistics

Name	*Synonyms*	
`minimum`	`min`	
`maximum`	`max`	
`sum`	`-`	
`mean`	`average, avg`	
`median`	`med`	
`standard_deviation`	`sd`	
`variance`	`var`	

3. Usage

3.1. Semantic query examples

- Value of a variable

  `@expr.origin`

- Call chain

  `@fun.(called_by)+` or

  `@fun.(calls)+`

- Side effect

  `mods.funs.dirty`

3.2. Semantic query examples

- Value of a variable

  `@expr.origin`

- Call chain

  `@fun.(called_by)+` or

  `@fun.(calls)+`

- Side effect

  `mods.funs.dirty`

- Function references

  `@fun.refs`

  `mods.funs[name=f].refs`

- Dynamic function references

Created by XMLmind XSL-FO Converter.

```
@expr.dynfun
```

```
@fun.dyncall
```

```
@fun.dyncalled_by
```

3.3. Exercises

- Find every string expression containing the “Error” substring!
- Find the definition and the references of the function `mnesia_log:open_log/4`!
- Find the function that calls the function `mnesia_log:open_log/4` outside from its defining module!
- Find the record definitions and usages!
- Find the variable binding points (expressions) for the variable `Log`!

Created by XMLmind XSL-FO Converter.

Chapter 20. Practice 6

1. Reminder

1.1. Syntax of the Semantic Queries

```
 <semantic query> ::=
    <initial selector> ('['<filter>']')* ('.' <subquery>)*
    ['.' <property> [':' <statistics>]]

<subquery> ::=
    <selector> ('['<filter>']')* |
    <iteration> ('['<filter>']')* |
    <closure> ('['<filter>']')*

<iterĂĄciĂł> ::=
    '{' <subquery> ('.' <subquery>)* '}' int

<closure> ::=
    '(' <subquery> ('.' <subquery>)* ')' int |
    '(' <subquery> ('.' <subquery>)* ')' '+'
```

1.2. Semantic Queries

- Basic entities: module, function, function clause, record, record field, variable, expression, macro
- Query = “sequence of filtered selectors | iterators | closures | properties“
- Each entity has a predefined set of selectors
- Each selector can be filtered based on its property
- Embedded queries are special selectors
- Global and position based (@) queries
- http://pnyf.inf.elte.hu/trac/refactorerl/

  wiki/SemanticQuery/Components

1.3. Semantic Query Examples

- `ri:q(mods.funs).`
- `ri:q(mods[name=mnesia_log].funs).`
- `ri:q(mods[name=mnesia_log].funs.name).`
- `ri:q(mods[name=mnesia_log].funs[.refs]).`
- `ri:q(mods[name=mnesia_log].funs[not .refs]).`

2. Use Cases

2.1. Finding Functions and References

Where is the function `mnesia_log:open_log/4` defined?

- `mods[name=mnesia_log].funs[name=open_log]`

- `mods[name=mnesia_log].funs[name=open_log, arity=4]`

Where is it referred?

- `mods[name=mnesia_log].funs[name=open_log].refs`
- or select a predefined query

2.2. Finding Functions and References

Which functions call `mnesia_log:open_log/4`?

-

```
mods[name=mnesia_log]
.funs[name=open_log].called_by
```

- `@fun.called_by`

Which functions are called from `mnesia_log:open_log/4`?

- `mods[name=mnesia_log].funs[name=open_log].calls`
- `@fun.calls`

2.3. Finding Records, Record Fields and References

Where is the record `mnesia_select` defined?

- `mods.records[name=mnesia_select]`
- `files.records[name=mnesia_select]`

Where is it referred?

- `mods.records[name=mnesia_select].refs`
- `@record.refs`
- or select a predefined queries

Where is the field `orig` referred?

-

```
  mods.records[name=mnesia_select]
.field[name=orig].refs
```

- `@field.refs`
- or select a predefined queries

2.4. Atom references

Where is the atom `mnesia_tid_locks` referred?

-

```
mods
 .funs
     .exprs
         .sub[type=atom, value=mnesia_tid_locks]
```

- Finding ets/dets/mnesia table references etc., process name references

2.5. String References

Where is the string `"Error message..."` used in the source code?

-

```
mods
.funs
    .exprs
        .sub[type=string,
             value~"Error message.*"]
```

2.6. An Advanced Query for Records

Is there any tuple which refers to the record backup_args?

-

```
 mods[name=mnesia_log]
.funs
    .exprs.sub[type=tuple,
               [.sub[index=1]
                    .origin[type=atom,
                            value =backup_args]]]
```

2.7. Detecting the Possible Values of a Variable

Where is the function `mnesia_log:open_log/4` referred?

- `mods[name=mnesia_log].funs[name=open_log].refs`

What is the value of the variable `Name`?

- `@expr.origin`
- %% or click on "Origin value”
- inter functional and inter modular

2.8. Detecting Dynamic Function Calls

- `ri:anal_dyn()`
- mods.funs.dynrefs
- Where is `mnesia:write/1` referred dynamically?
  -

```
mods[name=mnesia]
```


Created by XMLmind XSL-FO Converter.

```
                .funs[name=write, arity=1]
                   .dynrefs
```

  - `@fun.dynrefs`

- What are the functions called from `mnesia:safe_apply/1` dynamically?

  - `@fun.dynrefs`

2.9. Defining "Dynamic" Function References

```
     debug:safe_run(Par1,
             {CallbackModule, CallbackArgument},
             {Module, Function, [Arguments]}).



   {debug,                                         %% module
     safe_run,                                      %% function
     ['_', {'$1', '$2'}, {'$3', '$4', '$5'}],       %% match
     [
      {{'$1', '$2'} ,  {'$1', run, '$2'}},          %% mapping1
      {{'$3', '$4', '$5'} ,  {'$3', '$4', '$5'}}    %% mapping2
     ]
    }.
```

2.10. Defining "Dynamic" Function References

```
     gen_server:start(SrvName, Module, Args, Opt).



   {gen_server,                      %% module
    start,                           %% function
    ['_', '$1', '$2', '_'],          %% match
    [
     {'$1',   {'$1', init, '$2'}}    %% mapping1
    ]
   }.
```

2.11. Finding function calls

Which expressions call `mnesia_log:open_log/4` with an expression as a first parameter which value is `decision_log`?

-

```
            mods[name=mnesia_log]
           .funs[name=open_log]
              .refs
                  [.param[index=1]
                       .origin[type=atom,
                               value=decision_log]]
```

2.12. Finding function calls

Which modules calls `mnesia:foldr/4`? `mnesia`?

-

Created by XMLmind XSL-FO Converter.

```
mods[name=mnesia]
.funs[name=foldr]
     .called_by
```

Which functions calls `mnesia:foldr/4` outside from the module `mnesia`?

- ```
  mods[name=mnesia]
  .funs[name=foldr]
       .called_by[mod /= mnesia]
  ```

2.13. Calculating Macro Values

- `mods.macros`
- `mods.macro[name=DEBUG_TAB].refs`
- `@macro.refs`
- `@expr.macro_value`

Created by XMLmind XSL-FO Converter.
  ```

Chapter 21. Practice 7

1. Structural Complexity Metrics

1.1. Complexity metrics

- Large code base ⇛
  - high complexity
  - hard to identify weakness of the program
  - hard to measure development costs
- Metrics can help!

1.2. Metrics In RefactorErl

- Measure the structural complexity of Erlang source code
- Two ways to calculate them:
  - metric query language:

    `show max_depth_of_calling for module ('dyn')`
  - semantic query language:

    `mods[name==dyn].max_depth_of_calling`
- Automatic warning generation

1.3. Metric Query Language Examples

- `show number_of_fun for module ('a')`
- `show number_of_fun for module ('a','b')`
- `show branches_of_recursion for function ('a','f',1)`
- `show branches_of_recursion for function ('a','f',1,'a','g',0) sum`

1.4. Metric Query Language Examples

- `show metric for entity [aggregation filters]`
- entities: `module, function`
- aggregation filters: `max, avg, min, sum, fmaxname, totext, tolist`
- metrics: http://pnyf.inf.elte.hu/trac/refactorerl/wiki/MetricQuery
- `show loc for module ('a')`

2. Metrics as Semantic Query Properties

2.1. Software Metrics

Structural complexity metrics as properties

Created by XMLmind XSL-FO Converter.

- `mods.funs.is_tail_rec`
- `mods.funs.loc`
- `mods.funs.max_depth_of_cases`
- `mods.funs.branches_of_recursion`
- `mods.num_of_functions`

2.2. File Metrics

Name	*Synonyms*	*Type*
`module_sum`	`mod_sum`	int
`line_of_code`	`loc`	int
`char_of_code`	`choc`	int
`number_of_functions`	`num_of_fun, funnum`	int
`number_of_macros`	`num_of_mac, macnum`	int
`number_of_records`	`num_of_rec, recnum`	int
`included_files`	`inc_files`	int
`imported_modules`	`impmods`	int
`function_calls_in`	`callsin`	int
`function_calls_out`	`callsout`	int
`otp_used`	`otp`	int
`max_application_depth`	`maxappdepth`	int
`max_depth_of_calling`	`maxcalldepth`	int
`min_depth_of_calling`	`mincalldepth`	int
`max_depth_of_cases`	`maxcasedepth`	int
`max_depth_of_structures`	`maxstrdepth`	int
`number_of_funclauses`	`funclnum`	int
`branches_of_recursion`	`recbranches`	int
`mcCabe`	`mccabe`	int
`number_of_funexpr`	`funexprnum`	int

Created by XMLmind XSL-FO Converter.

number_of_messpass	messpassnum	int
max_length_of_line	maxlinelength	int
average_length_of_line	avglinelength	int
no_space_after_comma	-	int

2.3. Function Metrics

Name	*Synonyms*	*Type*
line_of_code	loc	int
char_of_code	choc	int
function_sum	fun_sum	int
max_application_depth	max_app_depth	int
max_depth_of_calling	maxcalldepth	int
max_depth_of_cases	maxcasedepth	int
max_depth_of_structures	maxstrdepth	int
number_of_funclauses	funclnum	int
branches_of_recursion		int
mcCabe	mccabe	int
calls_for_function	funcallin	int
calls_from_function	funcallout	int
number_of_funexpr		int
number_of_messpass		int
max_length_of_line	maxlinelength	int
average_length_of_line	avglinelength	int
no_space_after_comma		int
is_tail_recursive		int

2.4. Function Clause Metrics

Created by XMLmind XSL-FO Converter.

Name	*Synonyms*	*Type*
`line_of_code`	`loc`	int
`char_of_code`	`choc`	int
`function_sum`	`fun_sum`	int
`max_application_depth`	`max_app_depth`	int
`max_depth_of_calling`	`maxcalldepth`	int
`max_depth_of_cases`	`maxcasedepth`	int
`max_depth_of_structures`	`maxstrdepth`	int
`number_of_funclauses`	`funclnum`	int
`branches_of_recursion`		int
`mcCabe`	`mccabe`	int
`calls_for_function`	`funcallin`	int
`calls_from_function`	`funcallout`	int
`number_of_funexpr`		int
`number_of_messpass`		int
`max_length_of_line`	`maxlinelength`	int
`average_length_of_line`	`avglinelength`	int
`no_space_after_comma`		int
`is_tail_recursive`		int

3. Checking Coding Conventions

3.1. Coding Convention Rules

- Rule1: A module should not contain more than 400 lines

```
mods[line_of_code > 400]
```

```
mods.funs[line_of_code > 20]
```

- Rule2: Do not write deeply nested code (at most two levels)

```
@file.funs[max_depth_of_cases > 2]
```

```
mods[max_depth_of_cases > 2]
```


Created by XMLmind XSL-FO Converter.

- Rule3: Use no more than 80 characters in a line

```
mods.funs[max_length_of_line > 80]
```

- Rule4: Use space after commas

```
mods.funs[no_space_after_comma > 0]
```

- Rule5: Every recursive function should be tail recursive

```
mods.funs[is_tail_recursive==non_tail_rec]
```

4. Metric Mode

4.1. Metric Mode

- Automatic checking of rule violations
- Rules defined in the file `metricmod.defs`:

```
    {module_metrics,
 [{line_of_code,{100,1000}},
  {number_of_fun,{0,10}},
  ...]}.
{function_metrics,
 [{line_of_code,{0,20}},
  {char_of_code,{0,600}},
  ...]}.
```

-

```
ri:metricmode(on).
ri:metricmode(show).
ri:metricmode(off).
```

5. Exercises

5.1. Build Queries!

- Find modules that are using more than five macros!
- Find modules that have more than five function that have more than 5 return points.
- Find the functions in module `mnesia_log` that have more than five function clauses and are not tail recursive.
- Find the OTP callback modules!

Created by XMLmind XSL-FO Converter.

Chapter 22. Practice 8

1. Dependency Analysis

1.1. Dependency analysis

- Extracts information from the SPG
- Creates different views of the source code
- Shows the relations among different source code fragments

1.2. Types of dependency analysis

- Cyclic dependency detection
- Function/module dependency visualisation
- “Function block” relation visualisation
- Checking module relation based layers

1.3. Module dependencies

- Module A depends on module B ($A \longrightarrow B$) if there is at least one function call from A to B
- There is a cyclic dependency among modules, if there is a chain of dependencies: $A \longrightarrow B$, $B \longrightarrow C$, $\ldots$, $Y \longrightarrow Z$, $Z \longrightarrow A$.

1.4. Function dependencies

- Function A depends on function B ($A \Longrightarrow B$), if A calls B
- There is a cyclic dependency in the function dependency graph, if there is a chain of dependencies: $A \Longrightarrow B$, $B \Longrightarrow C$, $\ldots$, $Y \Longrightarrow Z$, $Z \Longrightarrow A$.

1.5. “Function-block” dependencies

- A “Function-block” is a set of related modules
- Can be related to “some functionality”
- This relation can be defined by the user, or the tool can use some predefined rules to calculate them
- Function-block A depends on function-block B ($A \hookrightarrow B$), if a module from A depends on a module from B

2. Usage

2.1. Function/module dependency analysis in ri

- `ri:draw_dep([{Key, Value}])` - draw a dependency graph using *Graphviz*
- `ri:print_dep([{Key, Value}])` - print the dependencies to the standard output

2.2. Parameters

- `{level, Level}` - the level of the analysis

Created by XMLmind XSL-FO Converter.

  - `func` - for function dependencies
  - `mod` - for module dependencies
- `{type, Type}`
  - `all` - the result contains graph nodes
  - `cycles` - the result contains the cyclic subgraph
  - `dep` - the result contains the whole graph as dependencies represented in list
- `{otp, Flag}`
  - `true` - when the Erlang/OTP standard modules must included in the result
  - `false` - otherwise list of values

2.3. Parameters

- `{gnode, List}` - list of entity or entities that should be the starting point of the analysis
  - `['mymod:myfunc/0']`
  - `[mymod]`
- `{exception, List}` -List of entities excluded from the analysis
  - `['mymod:myfunc/0']`
  - `[mymod]`
- `{leaves, List}` -list of those entities which should be included in the analysis, but their children should not (and consequently the children become exceptions)
  - `['mymod:myfunc/0']`
  - `[mymod]`
- `{dot, Path}` - The file path of the generated .dot graph description. Unless it is a non-existing absolute path, the graph will be placed into the `./dep_files directory`. This option is only available when using `draw_dep`.
  - `"/home/username/dir_of_deps"`

2.4. Smart graph

- The result of the smart graph generation is a html file: `ri:generate_smart_graph([{Key, Value}])`
- `{dependency_level, Level}` - the level of the analysis
  - `func` - for function dependencies
  - `mod` - for module dependencies
- `{output_path, Path}` - location and name of the generated .html file. If this option is not set, the name of the output file will be randomly generated, and the .html file will be placed in your tool/data directory.
  - `Path = string()`
- `dependency_options` - options of the dependency analysis.

Created by XMLmind XSL-FO Converter.

2.5. Examples

-

```
ri:print_dep([{level, mod}, {type, dep}]).
```

-

```
ri:print_dep([{level, func}, {type, cycles}]).

  [['foo:fv4/1','foo:fv4/1'],
  ['test3:p/1','test:fv6/1','test3:p/1'],
  ['cycle4:f4/1','cycle3:f3/1','cycle4:f4/1'],
  ['cycle2:fv2/1','cycle1:fv1/0','cycle2:fv2/1'],
  ['test:fv5/1','test:fv4/2','test:fv5/1'],
  ['cycle4:f5/1','cycle3:f6/1','cycle4:f5/1']]
```

-

```
ri:print_dep([{level, func}, {gnode, ["cycle4:f5/1"]}]).
```

-

```
DepOpts = [{level, mod},{type, all},{gnode, example_mod},
           {exception, []}, {leaves, []}, {otp, false}],
ri:generate_smart_graph([{dependency_level, mod},
                         {output_path, "/tmp/ex.html"},
                         {dependency_options, DepOpts}]).
```

2.6. Function-block analysis in ri

- `ri:fb_relations([{Key, Value}])`
- `{command, Command}`
  - `get_rel` - displays the relationship between the given function block list. The result is a tuplelist where a tuple represents a relation.
  - `is_rel` -decides whether there is a connection between the two given function blocks.
  - `check_cycle` -checks for cycles in the dependencies between the given function block list. Unless list is given, checks among every function block list.
  - `draw` - prints out the entire graph or creates a subgraph drawing from the given function block list. Output file is `fb_relations.dot`.
  - `draw_cycle` -prints out a subgraph which contains the cycle(s). Unless cycles exist, prints out the entire graph. Output file is `fb_rel_cycles.dot`.

2.7. Function-block analysis in ri

- `ri:fb_relations([{Key, Value}])`
- `{fb_list, List}`
  -

```
List = [string()] |
        [{Basename::string(),
          [Function block::atom()]}]
```


Created by XMLmind XSL-FO Converter.

- choose function block lists for further examinations. If no list given, then it takes every function block list, which means that every absolute path defines a function block.

- `{other, Other}`
  - `true` - when the category "Other" would be taken into consideration or not (true/false). The default value is true.
  - `false` - otherwise
  - The Other category is a special collector name for those modules which cannot be categorized into any function block. Practically this covers those modules which do not have directories (for example, usually `erlang`).

2.8. Examples

-
```
ri:fb_relations([{command, check_cycle}]).
```

-
```
ri:fb_relations([{command, is_rel},
                 {fb_list,
                  ["path_to_dir/subdir",
                   "path_to_dir/subdir/subsubdir"]}]).
```

-
```
ri:fb_relations([{command, is_rel},
                 {fb_list,{"path_to_dir", [1, 2]}}]).
```

2.9. Function-block analysis in ri

- `ri:fb_regexp([{Key, Value}])`
- `{type, Type}`
  - `list` - prints out every function block which matches the basic regular expression.
  - `get_rel` - decides whether there is a connection between the two given function blocks.
  - `cycle` - checks for cycles in the dependencies between the given function block list.
  - `draw` - prints out the entire graph or creates a subgraph drawing from the given function block list. Output file is `fb_relations.dot` or can be user defined with the dot key.

2.10. Function-block analysis in ri

- `ri:fb_regexp([{Key, Value}])`
- `{regexp, Value}`
  -
```
Value = [File::string() |
        [RegExp::string()]
```

If the regular expression is given in a file then every single regexp has to be defined in a separate line and must follow the Perl syntax and semantics as the http://www.erlang.org/doc/man/re.html erlang module resembles so. However, the user can give the regular expressions in a list as well.

2.11. Examples

```
 ri:fb_regexp([{type, list},
              {regexp, ["/usr/[a-zA-Z./]*/.*_syntax.*.erl"]}]).

Matched modules: [{"/usr/[a-zA-Z./]*/.*_syntax.*.erl",
                   [erl_syntax,erl_syntax_lib]}]
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
[{"/usr/[a-zA-Z./]*/.*_syntax.*.erl",
  [{'$gn',module,7},{'$gn',module,23}]}]
```

2.12. Examples

```
 ri:fb_regexp([{type, check_cycle},
              {regexp, ["/usr/[a-zA-Z./]*/.*_syntax.*.erl",
                        "/home/[a-zA-Z./]*/cycles"]}]).

Matched modules: [{"/usr/[a-zA-Z./]*/.*_syntax.*.erl",
                   [{'$gn',module,7},{'$gn',module,23}]},
                  {"/home/[a-zA-Z./]*/cycles",
                   [{'$gn',module,44},
                    {'$gn',module,43},
                    {'$gn',module,39},
                    {'$gn',module,40}]}]

Earlier results deleted (except .dot files and otp table).
Building "fb_rel" table...

[["/usr/local/lib/erlang/lib/syntax_tools-1.6.7.1/src",
  "/usr/local/lib/erlang/lib/syntax_tools-1.6.7.1/src"],
 ["/home/usrname/RefactorErl/test/cyclic/cycles",
  "/home/usrname/RefactorErl/test/cyclic/cycles"]]
```

2.13. Checking Layers in ri

- Groups of compilation units (in the case of Erlang, modules) usually form logical layers.
- Code in one layer should only use the layer immediately below it, and conversely, provide functionality only for the layer immediately above it.
- `ri:check_layered_arch/2,3` and `ri:show_layered_arch/2,3`

2.14. Checking Layers in ri

- `[atom(), [string()]]` - defines the name of the layer and the files contained by the layer (list them, path to them or regexp)
- `[{atom(), atom()}]` - list of allowed relations
- `srting()` - output file name

2.15. Examples

```
 ri:check_layered_arch(
   [ {il1,["^(/home/user/layers/layer1)$"]},
     {il2,["^(/home/user/layers/layer2)$"]},
```


Created by XMLmind XSL-FO Converter.

```
        {il3,["regexp3"]}],
      []).
  ri:show_layered_arch(
      [ {il1,["^(/home/user/layers/layer1)$"]},
        {il2,["^(/home/user/layers/layer2)$"]},
        {il3,["regexp3"]}],
      [{il1,il3}],"restrictions.dot").
```

2.16. Exercise

- Write your own view of the source code! (e.g. represent record dependency, macro dependency)
- Try to visualize it! You can use the existing components of RefactorErl

Created by XMLmind XSL-FO Converter.

Chapter 23. Practice 9

1. Clustering

1.1. Motivation

- We have a complex Erlang software
  - Consists of modules and functions
- Over time, it has grown to be even more complex
  - Maintenance became nearly impossible
- The modules should be grouped into blocks that are small enough to be maintained effectively

1.2. Clustering

- Code restructuring based on component relations
  - Function calls
  - Record and macro usage
- Module clustering
  - Split a large block of modules to more manageable parts
  - Involves splitting of header files
- Function clustering
  - Split a large module into smaller parts
  - Refactoring: move function

1.3. Types of clustering in RefactorErl

- Hierarchical algorithm (agglomerative)
  - In the beginning, each entity forms a separate cluster. Then, in each step, the two closest clusters are selected and unified. This process continues until there is only one cluster. The intermediate states contain a possible clustering of the entities. The output of the algorithm is the list of these possible clusterings.
- Genetic algorithm
  - Genetic algorithms simulate the evolution of species. There are iterations of populations in which every entity fights for survival or for the survival of its genes. A fitness function is defined to determine the value of an entity. The fitter an entity is, the more likely it survives. The algorithm is expected to converge to the fittest possible entity, like evolution does.

2. Usage in RefactorErl

2.1. Parameters for clustering

- Algorithm: The used clustering algorithm.
- Entities: The entities of the clustering. Modules and functions available for both algorithms.

Created by XMLmind XSL-FO Converter.

- Show Goodness: Yes/No question. If enabled, the tool shows the goodness values for each of the clusterings.
- Only best: Yes/No question. If enabled, the tool shows the best clustering result only.
- Store results: Yes/No question. If enabled, the tool stores the results in 3 different format describes bellow.

2.2. Output formats

- Dets table: It is used by the tool itself.
- Scriptable file: It is format which can easily be used by other programs. It is a list of pairs, every pair contains a keyword and a result.
- Readable file: It creates a report "readable to the human eye". It shows the resulting clusterings and the decomposition offered by the tool.

2.3. Parameters for agglomerative clustering

- Modules to skip: The list of module names (separated by space or comma characters) that should be ignored in the clustering process
- Functions to skip: The list of function names (separated by space or comma characters) that should be ignored in the clustering process
- Transform function: The function that transforms the attribute matrix before running the clustering. There are two options for the transformation: zero_one and none.
  - zero_one: The option zero_one means that the weights that are positive in the attribute matrix will be transformed to 1.
  - none: The option none means that no transformation will be performed in the attribute matrix.

2.4. Parameters for agglomerative clustering

- Distance function: It can be call_sum, weight or a function reference to user-defined function.
  - call_sum: Distance function based on function call structure, sums call weights.
  - weight: The distance function is based on function call structure and record usage. It is weighted by the anti-gravity factor.
  - User-defined function
- Anti-gravity: The anti-gravity factor for distance calculating function, like weight.

2.5. Parameters for agglomerative clustering

- Merge function: The cluster attribute calculator functions are used in the attribute matrix user algorithm. This function calculates the new attributes of the created clusters.
  - smart: The size attributes are summed, the entities attributes are merged, average is calculated from the function, record and macro attributes, and the other attributes are undefined.
  - User-defined function

2.6. Parameters for genetic clustering

- Population size: The number of chromosomes in every iteration of the algorithm. At the beginning of the algorithm a random population is generated.
- Iterations: The number of iteration in the algorithm. For default type 10.

Created by XMLmind XSL-FO Converter.

- Mutation rate: The probability of mutation. For default type 0.9.
- Crossover rate: The probability that a crossover will be performed on two selected chromosomes. For default type 0.7.

2.7. Parameters for genetic clustering

- Elite count: The number of chromosomes that are transferred to the next generation without change. For default type: 2
- Maximum cluster size: Maximum number of clusters allowed.
- Maximum start cluster size: Maximum number of clusters allowed at startup.

2.8. Parameters for decomposition

- Decomposition: It shows whether the user wants a possible decomposition of the modules or not. Only available with module clustering.
- Library limit: The minimum number of function calls for library modules. If a module is called by at least this many other modules, it is considered a library modules.
- Headers: The format of header files. It is a string which is matched to the end of the file names.

2.9. Running the clustering on Mnesia!

```
 ri:cluster().
Please choose an item from the list (blank to abort).

Please select the clustering algorithm
 1. Agglomerative
 2. Genetic
 type the index of your choice: g
Answer with an index in range
 type the index of your choice: 2
Please choose an item from the list (blank to abort).

Please select the clustering entities
 1. Module
 2. Function
 type the index of your choice: 1
Please choose an item from the list (blank to abort).

May I offer a decomposition of the modules?
 1. Yes
 2. No
 type the index of your choice: 1
```

2.10. Running the clustering on Mnesia!

```
 Please answer the following question (blank to abort).
 Choose population size 10
Please answer the following question (blank to abort).
 Choose iteration numbers 10
Please answer the following question (blank to abort).
 Choose mutation rate 0.9
Please answer the following question (blank to abort).
 Choose crossover rate 0.7
Please answer the following question (blank to abort).
 Choose elite count 2
Please answer the following question (blank to abort).
 Choose maximum cluster size 10
```


Created by XMLmind XSL-FO Converter.

```
Please answer the following question (blank to abort).
 Choose maximum stating cluster size 15
Please answer the following question (blank to abort).
 Minimum call number for library modules 20
Please answer the following question (blank to abort).
 Format of header file names "hrl"
```

2.11. Running the clustering on Mnesia!

```
 Please choose an item from the list (blank to abort).
May I show only the best clustering solution?
 1. Yes
 2. No
 type the index of your choice: 1
Please choose an item from the list (blank to abort).

Do you need the goodness values?
 1. Yes
 2. No
 type the index of your choice: 1
Please choose an item from the list (blank to abort).

Do you want to store the results?
 1. Yes
 2. No
 type the index of your choice: 1
```

2.12. Running the clustering on Mnesia!

```
 See the direct information feed below:
 =========================
===Clustering results:===
=========================
 [mnesia_tm,mnesia_schema,mnesia_locker]
 [mnesia_sp,mnesia_monitor,mnesia_index]
 [mnesia_snmp_sup,mnesia_kernel_sup,mnesia_checkpoint_sup,mnesia_backup]
 [mnesia_subscr,mnesia_registry,mnesia_controller,df]
 [mnesia_recover,mnesia_log,mnesia_frag_old_hash,mnesia]
 [mnesia_snmp_hook,mnesia_dumper]
 [mnesia_loader,mnesia_late_loader,mnesia_frag,mnesia_event,mnesia_bup]
 [mnesia_sup,mnesia_lib,mnesia_checkpoint]
 [mnesia_text,mnesia_frag_hash]
 === Fitness Numbers: ===
 1.5323693231184643
 === Decomposition: ===
 Filename: /usr/lib/erlang/lib/mnesia-4.8/src/mnesia.erl
     Can't move: [{fun_attr,mnesia,select_state,2},
                  {fun_attr,mnesia,snmp_order_keys,4},
                  ...
```

2.13. Exercise

- Running the agglomerative clustering on Mnesia!

Chapter 24. Practice 10

1. Refactoring

1.1. Refactoring with RefactorErl

- Platform for source code transformations – 24 implemented refactorings
  - Rename
  - Move definition
  - Expression structure
  - Function interface
  - Parallelisation

1.2. Refactoring steps

Rename

- variable
- function
- record, record field
- macro
- module/header file

Function interface

- introduce function argument
- reorder parameters
- introduce tuple
- eliminate/introduce import

1.3. Refactoring steps

Rename

- variable
- function
- record, record field
- macro
- module/header file

Function interface

- introduce function argument

Created by XMLmind XSL-FO Converter.

- reorder parameters
- introduce tuple
- eliminate/introduce import

Move definition

- macro
- record
- function

Expression structure

- eliminate/introduce variable
- eliminate/introduce function
- eliminate macro application
- eliminate fun-expression

Data structure

- Introduce record
- Upgrade module interface

2. Rename Refactorings

2.1. Rename Variable

- Renames a variable and all of its occurrences
- If the user does not specify a variable, the transformation starts an interaction to ask for a variable. It gives a list of variables which can be reached from the selected function clause.
- Side conditions:
  - The new variable name does not exist in the scope of the variable
- Transformations steps and compensations:
  - Replace every occurrence of the variable with the new name. In case of variable shadowing, other variables with the same name are not modified.

2.2. Rename X to Y

```
    max(X,Z) ->
     if
        X=<Z -> Z;
        X>=Z -> X
     end.
```

⇒

```
  max(Y,Z) ->
     if
        Y=<Z -> Z;
```


Created by XMLmind XSL-FO Converter.

```
        Y>=Z -> Y
    end.
```

2.3. Rename Function

- Renames a function
- If the user does not specify a function to be renamed or the specified function does not exist, the transformation starts an interaction to ask the user to specify one. The user has to select a function from a radio group.
- Side conditions:
  - There must be no function with the given name and the same arity as the function to be renamed among the functions in the module, the functions imported in the module, and the auto-imported BIFs.
  - There must be no function with the given name and the same arity as the function to be renamed among the local and imported functions in the modules that import the function to be renamed.
- Transformations steps and compensations

2.4. Rename Function

- Renames a function
- Side conditions
- Transformations steps and compensations:
  - The name label of the function is changed at every branch of the definition to the new one.
  - In every static call to the function, the old function name is changed to the new one.
  - Every implicit function expression is modified to contain the new function name instead of the old one.
  - If the function is exported from the module, the old name is removed from the export list and the new name is put in it.
  - If the function is imported in an other module, the import list is changed in that module to contain the new name instead of the old one.

2.5. Rename doit to send_start

```
   doit(P) ->
    P ! {msg, start}.

 start(L) ->
    lists:forall(
       fun doit/1, L).
```

⇒

Created by XMLmind XSL-FO Converter.

```
send_start(P) ->
   P ! {msg, start}.

start(L) ->
   lists:forall(
      fun send_start/1, L).
```

2.6. Rename Record

- Renames a record
- If one of the side conditions fails, the transformation starts an interaction to ask for a new record name.
- If the user does not specify a record, the transformation starts an interaction to ask the user to specify a record.
- Side conditions:
  - There must be no record with the new name
    - in the file that contains the record,
    - in files which are included by this file,
    - in files which include this file.
- Transformations steps and compensations:
  - The record name is changed to the new name in the definition of the record and in every record expression that refers the record.

2.7. Renaming record "person" to "member"

```
 -record(person, {name, age}).

rename(Arg, New) ->
   #person{name=Name} = Arg,
   io:format("%s", [Name]),
   Arg#person{name=New}.
```

⇒

```
-record(member, {name, age}).

 rename(Arg, New) ->
    #member{name=Name} = Arg,
    io:format("%s", [Name]),
    Arg#member{name=New}.
```

2.8.

- Renames a record files
- If the user does not specify a record field, then the transformation starts an interaction to ask the user to specify one.

Created by XMLmind XSL-FO Converter.

- Side conditions:
  - The record must have no field with the same name as the given new field name. If it has, the transformation starts an interaction to ask for a new record field name.
- Transformations steps and compensations:
  - The field name is changed to the new name in the definition of the record and in every record expression that refers the field.

2.9. Renaming field name to id

```
 -record(member, {name, age}).

rename(Arg, New) ->
   #member{name=Name} = Arg,
   io:format("%s == %s",
      [Name, Arg#member.name]),
   Arg#member{name=New}.
```

⇛

```
-record(member, {id, age}).

 rename(Arg, New) ->
    #member{id=Name} = Arg,
    io:format("%s == %s",
       [Name, Arg#member.id]),
    Arg#member{id=New}.
```

2.10.

- Renames a macro
- If one of the side conditions fails, the transformation starts an interaction to ask for a new macro name.
- If the user does not specify a macro or the specified macro does not exist, the transformation starts an interaction to ask for a macro.
- Side conditions:
  - No macro already exist with the same new name in either
    - in a file hosting definition or usage of the macro,
    - in files included by the said,
    - in files that include the said.
- Transformations steps and compensations:
  - The macro name is replaced with the new name at both definitions and all usage sites.

Created by XMLmind XSL-FO Converter.

2.11. Renaming LessEq to Leq

```
  -define(LessEq, =<).

max(X,Z) ->
   if
      X ?LessEq Z -> Z;
      X>=Z -> X
   end.
```

⇒

```
-define(Leq, =<).

max(Y,Z) ->
   if
      Y ?Leq Z -> Z;
      Y>=Z -> Y
   end.
```

2.12.

- Renames a header file
- If the second side condition fails, the transformation starts an interaction to ask for a new name
- Side conditions:
  - The type of the file has to be a header file. If the pointed file is a module, the transformation will fail.
  - The directory must not contain a file having the same name as new name given.
- Transformations steps and compensations:
  - Rename the header file name to the new name on the graph.
  - Rename the references to the header file in the include forms.
  - Rename or move and rename the file to the new name.

2.13. Renaming header file header1.hrl to newname

```
  %% header.hrl
-define(M, ok).

-define(Add(X, Y),
        X + Y).

%% refmod1.erl
-module(refmod1).
-include("header.hrl").

func1(A, B) ->
   ?Add(A, B).

func2() ->
```


Created by XMLmind XSL-FO Converter.

```
    ?M.
```

⇒

```
%% newname
-define(M, ok).

-define(Add(X, Y),
           X + Y).

%% refmod1.erl
-module(refmod1).
-include("newname").

func1(A, B) ->
   ?Add(A, B).

func2() ->
   ?M.
```

2.14. Rename module

- Renames a module
- If one of the side conditions fails, the transformation starts an interaction to ask for a new module name.
- Side conditions:
  - The given new name should be a legal file name.
  - There must not exist another module with the given new name in the graph.
  - There must not exist another file with the given new name in the directory of the module to be renamed.
- Transformations steps and compensations:
  - Rename the current module name to the new name.
  - Rename the collected module qualifiers to the given new name.
  - Rename the references to the module in the import lists.
  - Rename the file to the new name.

2.15. Renaming module mod1 to newmod

```
  -module(mod1).
-export([add/2]).

add(A,B) -> A + B.


-module(mod2).
-export([add/1]).

add([]) -> [];
add[{A,B}|Tail]) ->
   [mod1:add(A,B)
```


Created by XMLmind XSL-FO Converter.

```
     |add(Tail)].
```

⇒

```
-module(newmod).
-export([add/2]).

add(A,B) -> A + B.


-module(mod2).
-export([add/1]).

add([]) -> [];
add[{A,B}|Tail]) ->
   [newmod:add(A,B)
    |add(Tail)].
```

3. Function Interface

3.1. Introduce Function Parameter/Generalize Function

- Generalizes a function definition by selecting an expression (or an expression sequence), and makes this the value of a new parameter added to the definition of the function. The actual parameter at every call site becomes the selected part with the corresponding compensation.
- Side conditions:
  - The name of the function with its arity increased by one should not conflict with another function, either defined in the same module, imported from another module, or being an auto-imported built-in function.
  - The new variable name does not exist in the scope of the selected expression(s) and must be a legal variable name. If the new variable name does not keep these conditions, the transformation starts an interaction to ask for a new variable name.
  - The starting and ending positions should delimit an expression or a sequence of expressions.

3.2. Introduce Function Parameter/Generalize Function

- Side conditions:
  - The selected expressions do not bind variables that are used outside the selection.
  - Variable names bound by the expressions do not exist in the scopes of the generalized function calls.
  - The expressions to generalize are not patterns and they do not call their containing function.
  - If the selection is part of a list comprehension, it must be a single expression and must not be the generator of the comprehension.
  - The extracted sequence of expressions are not part of a macro definition, and are not part of macro application parameters.
- Transformations steps and compensations:
  - http://pnyf.inf.elte.hu/trac/refactorerl/wiki/RefactoringSteps/

Created by XMLmind XSL-FO Converter.

GeneralizeFunction#Transformationstepsandcompensations

3.3. Introduce a new parameter to function double/1

```
  -module(gen1).
-export([double/1]).

double(List) ->
   [2*X || X <- List].

f(Z)->
   double(Z).
```

⇒

```
-module(gen1).
-export([double/1]).

double(List) ->
   double(List,2)

double(List, N) ->
   [N*X || X <- List]

f(Z)->
   double(Z, 2).
```

3.4. Reorder function parameters

- Reorder the parameters of the function
- When the given order is not legal, the transformation starts an interaction to ask for a new order.
- Side conditions:
  - When a function application has an argument with side effects, the transformation may only be carried out after a warning that the order of side effects most probably will change, which may change the way the program works.
- Transformations steps and compensations

3.5. Reorder function parameters

- Reorder the parameters of the function
- When the given order is not legal, the transformation starts an interaction to ask for a new order.
- Side conditions
- Transformations steps and compensations:
  - Change the order of patterns in every formal parameter list in the function according to the given new order.
  - For every static call of the function, change the order of the expressions that provide the actual parameters to the call according to the given order.

Created by XMLmind XSL-FO Converter.

- Every implicit function expression is expanded to the corresponding explicit expression which contains a static call to the reordered function.
- For every call of the function that provides the arguments as a list, insert a compensating function expression that changes the order of the elements in the list according to the given new order.

3.6. Reordering parameters

```
  sum(A,B,C) ->
   A+B+C.

caller() ->
   sum(1, 2, 3).
```

⟹

```
sum(C,B,A) ->
   A+B+C.

caller() ->
   sum(3, 2, 1).
```

3.7. Introduce tuple / Tuple function parameters

- Creates a tuple from the selected function arguments
- Side conditions:
  - The function must be declared at the top level of a module, not a function expression.
  - If the number of parameters that should be contracted into tuple is greater than one the arity of function will be changed. In this case the function with new arity should not conflict with other functions.
  - If the function is not exported, it should not conflict with other functions defined in the same module or imported from other modules.
  - If the function is exported, then besides the requirement above, for all modules where it is imported, it should not conflict with functions defined in those modules or imported by those modules.
- Transformations steps and compensations

3.8. Introduce tuple / Tuple function parameters

- Creates a tuple from the selected function arguments
- Side conditions
- Transformations steps and compensations:
  - Change the formal parameter list in every clause of the function: contract the formal arguments into a tuple pattern from the first to the last argument that should be contracted.
  - If the function is exported from the module, then the arity of the function is updated with new arity in the export list.
  - If the function is exported and another module imports it, then the arity must be adjusted in the import lists.
  - Implicit function references are turned into fun expressions that call the function, and the result is handled as any other function call.

Created by XMLmind XSL-FO Converter.

- For every application of the function, modify the actual parameter list by contracting the actual arguments into a tuple from the first to the last argument that should be contracted.

3.9. Create a tuple from the arguments of step/2

```
  step(A,B) ->
  {B, A rem B}.

gcd(A,B) ->
  if
    B==0 -> A;
    true ->
      {X,Y} = step(A,B),
      gcd(X,Y)
  end.
```

⇒

```
step({A,B}) ->
  {B, A rem B}.

gcd(A,B) ->
  if
    B==0 -> A;
    true ->
      {X,Y} = step(A,B),
      gcd(X,Y)
  end.
```

3.10. Introduce Import List Element

- Introduces (or eliminates) an import argument for a function.
- If any of the side conditions fails, the transformation asks for a new module. It gives a list to the user to specify a module. The list contains the modules from where the source module has already imported functions.
- Side conditions:
  - No local function of the file has the same name and arity as the functions of the module that are used or imported in the file.
  - No imported function in the file has the same name and arity as functions of the module that are used in the file.
- Transformations steps and compensations

3.11. Introduce Import List Element

- Introduces (or eliminates) an import argument for a function.
- If any of the side conditions fails, the transformation asks for a new module. It gives a list to the user to specify a module. The list contains the modules from where the source module has already imported functions.
- Side conditions
- Transformations steps and compensations:

Created by XMLmind XSL-FO Converter.

- In case there is no import list of the module in the file a new import list containing the functions of the module that are used in the file is added to the file.
- In case there is only one import list of the module in the file the rest of the functions used in the file are added to this list.
- In case there is more then one import list of the module, the contents of this list will be merged in one, and the rest of the functions used in the file are added to this list.
- The module qualifiers of the module are removed from the corresponding functions.

3.12. Introducing an import list for lists:sort/1

```
  -export([my_sort/1]).

my_sort(A)->
   lists:sort(A).
```

⇒

```
-export([my_sort/1]).

-import(lists, [sort/1]).

my_sort(A)->
   sort(A).
```

4. Move Definitions

4.1. Move macro

- Moves a macro definition from a file to the specified file
- If the user does not specify the macros to be moved, the transformation starts an interaction to ask the user to specify macros. The user has to select the macros to be moved from a checkbox list.
- Side conditions:
  - http://pnyf.inf.elte.hu/trac/refactorerl/wiki/RefactoringSteps/

    MoveMacro#Sideconditions
- Transformations steps and compensations:
  - http://pnyf.inf.elte.hu/trac/refactorerl/wiki/RefactoringSteps/

    MoveMacro#Transformationstepsandcompensations

4.2. Moving macro Person the header.hrl

```
  %%header.hrl
```


Created by XMLmind XSL-FO Converter.

```
-define(Ok, ok).

%%mm.erl

-module(mm).
-define(Person(Name,Age),
        {Name,Age}).

f() -> ?Person("John",33).
```

⇒

```
%%header.hrl

-define(Ok, ok).
-define(Person(Name,Age),
        {Name,Age}).

%%mm.erl

-module(mm).
-include("client.hrl").

f() -> ?Person("John",33).
```

4.3. Move record

- Moves a record definition from a file to the specified file
- If the user does not specify the records to be moved, the transformation starts an interaction to ask the user to specify records. The user has to select the records to be moved from a checkbox list.
- Side conditions:
    - http://pnyf.inf.elte.hu/trac/refactorerl/wiki/RefactoringSteps/

      MoveRecord#Sideconditions
- Transformations steps and compensations:
    - http://pnyf.inf.elte.hu/trac/refactorerl/wiki/RefactoringSteps/

      MoveRecord#Transformationstepsandcompensations

4.4. Moving record msg to message.hrl

```
 %%client.hrl

-record(conn, {ip, port=80}).
-record(msg, {sender, text}).

%%messages.hrl

-record(dmsg, {date, text}).

%%client.erl

-include("client.hrl").
```


Created by XMLmind XSL-FO Converter.

```
sendmessage(Msg) ->
   ?SERVER ! #msg{text=Msg}.
```

⇒

```
%%client.hrl

-record(conn, {ip, port=80}).

%%messages.hrl

-record(dmsg, {date, text}).
-record(msg, {sender, text}).

%%client.erl

-include("client.hrl").
-include("messages.hrl").

sendmessage(Msg) ->
   ?SERVER ! #msg{text=Msg}.
```

4.5. Move function

- Moving a function from a module to another module
- If the user do not select functions to be moved, the transformation starts an interaction. The tool gives a checkbox list to the user to select the functions to be moved.
- Side conditions:
  - http://pnyf.inf.elte.hu/trac/refactorerl/wiki/RefactoringSteps/

    MoveFunction#Sideconditions
- Transformations steps and compensations:
  - http://pnyf.inf.elte.hu/trac/refactorerl/wiki/RefactoringSteps/

    MoveFunction#Transformationstepsandcompensations

4.6. Moving pzip/1 to xlists.erl

```
 %%from.erl
-module(from).
-export([print/1,pzip/1]).

print(Pairs) ->
   io:format("~p~n",
      pzip(Pairs)).

pzip([A,B|Rest]) ->
   [{A,B}|pzip(Rest)];
pzip(_) ->
   [].

%%xlists.erl
-module(xlists).
-export([flatsort/1]).
```


Created by XMLmind XSL-FO Converter.

```
flatsort(Xs) ->
   lists:usort(
      lists:flatten(Xs)).
```

⇒

```
%%from.erl
-module(from).
-export([print/1]).

print(Pairs) ->
   io:format("~p~n",
      xlists:pzip(Pairs)).

%%xlists.erl
-module(xlists).
-export([flatsort/1]).
-export([pzip/1]).

flatsort(Xs) ->
   lists:usort(
      lists:flatten(Xs)).

pzip([A,B|Rest]) ->
   [{A,B}|pzip(Rest)];
pzip(_) ->
   [].
```

5. Data Structure Related Refactorings

5.1. Introduce record

- Introduces a record instead of a tuple function parameter
- If the name or the field name is not legal, the transformation starts an interaction to ask for a new name.
- Side conditions:
  - http://pnyf.inf.elte.hu/trac/refactorerl/wiki/RefactoringSteps/

    IntroduceRecord#Sideconditions
- Transformations steps and compensations:
  - http://pnyf.inf.elte.hu/trac/refactorerl/wiki/RefactoringSteps/

    IntroduceRecord#Transformationstepsandcompensations

5.2. Introducing the record cart

```
 -export([mul/2]).

mul({Re1,Im1},{Re2,Im2})->
   {Re1*Re2-Im1*Im2,
   Re1*Im2+Im1*Re2}.
```

⇒

```
-export([mul/2]).
```


Created by XMLmind XSL-FO Converter.

```
-record(cart, {re, im}).

mul(#cart{re=Re1, im=Im1},
    #cart{re=Re2, im=Im2})->
   #cart{re=Re1*Re2-Im1*Im2,
   im=Re1*Im2+Im1*Re2}.
```

5.3.

•

- Side conditions:
  - http://pnyf.inf.elte.hu/trac/refactorerl/wiki/RefactoringSteps/

    UpgradeModuleInterface#Sideconditions
- Transformations steps and compensations:
  - http://pnyf.inf.elte.hu/trac/refactorerl/wiki/RefactoringSteps/

    UpgradeModuleInterface#Transformationstepsandcompensations

5.4. Upgrading regexp:match/2

```
 case regexp:match(String, RE) of
   {error, Reason} -> throw("Error: " ++ lists:flatten(
                        regexp:format_error(Reason)));
   {match, 1, Len} -> Len;
   {match, St, Len} -> use(St, Len);
   nomatch -> ok
end
```

⇒

```
try re:run(String, RE, [{capture, first}]) of
   {match, [{0, Len}]} -> Len;      % [{1-1, Len}] works too
   {match, [{St, Len}]} -> use(St+1, Len);
   nomatch -> ok
catch
   error:badarg -> throw("Error: " ++ lists:flatten(
                     "Bad regular expression"))
end
```

6. Expression Structure

6.1. Eliminate Variable

- All instances of a variable are replaced with its bound value.
- If the selection is not specify a variable but it is inside a function clause, the tool gives a list to the user to select a variable. The list contains the reachable variables in the given function clause.

Created by XMLmind XSL-FO Converter.

- Side conditions:
  - The variable has exactly one binding occurrence on the left hand side of a pattern matching expression, and not a part of a compound pattern.
  - The expression bound to the variable has no side effects.
  - Every variable of the expression is visible (that is, not shadowed) at every occurrence of the variable to be eliminated.
- Transformations steps and compensations

6.2. Eliminate Variable

- All instances of a variable are replaced with its bound value.
- If the selection is not specify a variable but it is inside a function clause, the tool gives a list to the user to select a variable. The list contains the reachable variables in the given function clause.
- Side conditions
- Transformations steps and compensations:
  - Every occurrence of the variable is substituted with the expression bound to it at its binding occurrence, with parentheses around the.
  - If the result of the match expression that binds the variable is discarded, the whole match expression is removed. Otherwise, the match expression is replaced with its right hand side.

6.3. Eliminating the variable Y

```
func(X) ->
 Y=X+2,
 Pid ! {value,Y},
 Y.
```

⇒

```
func(X) ->

   Pid ! {value,X+2},
   X+2.
```

6.4. Introduce variable/Merge subexpression duplicates

- A new match expression is created that binds the selected expression to the variable that the user has given as input, and all instances of the expression is changed to the variable.
- If the given variable name is not legal then the transformation starts an interaction to ask for a new variable name.

Created by XMLmind XSL-FO Converter.

- Side conditions:
  - http://pnyf.inf.elte.hu/trac/refactorerl/wiki/RefactoringSteps/

    MergeExpressions#Sideconditions
- Transformations steps and compensations:
  - http://pnyf.inf.elte.hu/trac/refactorerl/wiki/RefactoringSteps/

    MergeExpressions#Transformationstepsandcompensations

6.5. Introducing variable V

```
 foo(A,B) ->

  peer ! {note, A+B},
  A+B.
```

⇒

```
foo(A,B) ->
   V = A+B,
   peer ! {note, V},
   V.
```

6.6. Inline Function

- Substitutes the selected application with the corresponding function body and executes compensations
- If the user does not specify a function application or the specified function does not exist, the transformation starts an interaction to ask the user to specify one. The user has to select a function from a radio group.
- Side conditions:
  - http://pnyf.inf.elte.hu/trac/refactorerl/wiki/RefactoringSteps/

    InlineFunction#Sideconditions
- Transformations steps and compensations:
  - http://pnyf.inf.elte.hu/trac/refactorerl/wiki/RefactoringSteps/

    InlineFunction#Transformationstepsandcompensations

6.7. Inlining sort/1

```
 -module(in).

-export([sort/2]).

sort(A, B) ->
```


Created by XMLmind XSL-FO Converter.

```
      sort({A, B}).

   sort({A, B}) when A > B ->
      {A, B};
   sort({A, B}) ->
      {B, A}.
```

⇛

```
  -module(in).

   -export([sort/2]).

   sort(A, B) ->
      case {A, B} of
         {A, B} when A > B ->
            {A, B};
         {A, B} ->
            {B, A}
      end.

   sort({A, B}) when A > B ->
      {A, B};
   sort({A, B}) -> {B, A}.
```

6.8. Introduce function/Extract function

- A function definition might contain an expression or a sequence of expressions which can be considered as a logical unit, hence a function definition can be created from it. The extracted function is lifted to the module level, and it is parameterized with the free variables of the original expressions: those variables which are bound outside of the expressions, but the value of which is used by the expressions.
- Side conditions:
  - http://pnyf.inf.elte.hu/trac/refactorerl/wiki/RefactoringSteps/

    ExtractFunction#Sideconditions
- Transformations steps and compensations:
  - http://pnyf.inf.elte.hu/trac/refactorerl/wiki/RefactoringSteps/

    ExtractFunction#Transformationstepsandcompensations

```

```

6.9. Introducing two_sol/3

```
    -module(quadratic).

   -export([f/0]).

   solve(A,B,C) ->
      D := B * B - 4 * A * C,
      if
         D == 0 ->
            {-B / 2 / A};
         D > 0 ->
            S = math:sqrt(D),
            {-(B+S)/2/A,
            -(B-S)/2/A}.
         D < 0 ->
```


Created by XMLmind XSL-FO Converter.

```
            no_solution
      end.
```

⇛

```
   -module(quadratic).

    -export([f/0]).

    solve(A,B,C) ->
       D := B * B - 4 * A * C,
       if
          D == 0 ->
             {-B / 2 / A};
          D > 0 ->
             two_sol(A, B, D);
          D < 0 ->
             no_solution
       end.

    two_sol(A, B, D) ->
       Sqrt = math:sqrt(D),
       {-(B+S)/2/A,
        -(B-S)/2/A}.
```

6.10. Inline macro

- Substitutes a selected macro application with the corresponding macro body
- If the selection is not specify a macro usage the transformation starts an interaction to let the user specify one. It gives a list with the possible macro usages.
- Side conditions:
  - http://pnyf.inf.elte.hu/trac/refactorerl/wiki/RefactoringSteps/

    InlineMacro#Sideconditions
- Transformations steps and compensations:
  - http://pnyf.inf.elte.hu/trac/refactorerl/wiki/RefactoringSteps/

    InlineMacro#Transformationstepsandcompensations

6.11. Inlining ?Add(A,A)

```
     -module(inlmac).
   -define(Add(A,B),A+B).

   double(A)-> ?Add(A,A).
```

⇛

```
   -module(inlmac).
   -define(Add(A,B),A+B).
```


Created by XMLmind XSL-FO Converter.

```
double(A)->A+A.
```

6.12. Eliminate fun expression/Expand fun expression

- Expands an implicit fun expression into an explicit one
- Side conditions:
  - The selected expression should be an implicit fun expression or part/subexpression of an implicit fun expression.
- Transformations steps and compensations:
  - If the implicit fun expression is found, the new syntax structure is created and the old expression is replaced with the new one.

6.13. Eliminating implicit reference to far:away/2

```
  -module(near).

-export([f/0]).

f() ->
   fun far:away/2.
```

⇒

```
-module(near).

-export([f/0]).

f() ->
   fun(V1, V2) ->
      far:away(V1, V2)
   end.
```

6.14. Introduce/eliminate list comprehensions

- Turns lists:map, lists:foreach and lists:filter calls into list comprehension syntax, or do it backwards.
- Side conditions:
  - http://pnyf.inf.elte.hu/trac/refactorerl/wiki/RefactoringSteps/

    ListComprehensions#Sideconditions
- Transformations steps and compensations:
  - http://pnyf.inf.elte.hu/trac/refactorerl/wiki/RefactoringSteps/

    ListComprehensions#Transformationstepsandcompensations

Created by XMLmind XSL-FO Converter.

6.15. Transforming to lists:filter/2

```
f(Xs) ->
 [X || X <- Xs,
       X < 5].
```

⇒

```
f(Xs) ->
   lists:filter(
      fun(X) ->
         X < 5
      end, Xs).
```

6.16. Exercise

- Try out the refactorings in Emacs!
- Try out the refactorings using ri/ris!
- Try out the refactorings in Vi, Eclipse!
- Is there any difference?

Created by XMLmind XSL-FO Converter.

Chapter 25. Practice 11

1. Refactoring with RefactorErl

1.1. Refactoring workflow

1. Read and analyse source code

   - Already finished when refactoring starts

2. Check side conditions

   - Semantic links make it easy and efficient
   - Graph queries simplify graph traversal

3. Apply the transformation

   - Syntax tree based manipulations

4. Save the result

   - Unmodified code is preserved
   - Generated and moved code is pretty printed

1.2. Transformation

- Only the syntax tree is manipulated
  - Syntactic nodes can be created or deleted
  - Subtrees can be copied or moved
- Automatic token handling
  - Missing or extra commas and semicolons
  - Generation or removal based on the syntax description
- Automatic analysis
  - Incremental semantic analysis is triggered by syntactic changes
  - Pretty printing is a special kind of analysis

1.3. Implementation

- Every refactoring implementation must export the function `prepare/1`
- The transformation framework calls this function

Created by XMLmind XSL-FO Converter.

- The return value of the function is a list of fun expressions implementing the manipulation of the syntax tree: the transformation and compensation steps.
- Refactorings may throw `?RefError(Condition, Details)` and `?LocalError(Condition, Details)` error messages.

1.4. prepare/1

- Collects the parameters of the refactoring using the module `?Args`
  - it will return the asked refactoring parameters calculated from the `ArgList` parameter of the refactoring (e.g. ranges of expressions, names)
  - or ask the user the specify the required parameters (interaction)

1.5. prepare/1

- Collects the information that is necessary to check the side conditions of the transformations
- Checks the side conditions
- Asks the user for extra input when required, for example
  - when the introduced new variable name may conflict with existing names, the transformations ask a new name
- Collects all information that is required to perform the transformation and the compensation
- Creates the fun expressions containing the AST manipulations

1.6. Transformations in general

- removal of comments from all transformed parts
- syntax tree manipulations
- reinsertion of the removed comments

1.7. Restrictions

- Once an AST modification(insert, remove, update) started, it is not possible to query information from the graph
- Thus all necessary information gathering must be presented at the beginning of each fun-expression

1.8. Error Messages

- `?RefError(Condition, Details)` or `?LocalError(Condition, Details)`
- `Condition` is an atom, and generally states the nature of the problem (e.g. `side_eff`).
- `Details` is a list that has information relevant to the error.
- If there are no details, use `?RefErr0r(Condition)` and `?LocalErr0r(Condition)`. These put empty lists in Details.
- `?LocalErrors` are processed by the transformation module's `error_text/2` function.
- `?RefErrors` are processed by `?Error:error_text/2`.

Created by XMLmind XSL-FO Converter.

2. Case Study: Rename Variable

2.1. Rename Variable

- Renames a variable and all of its occurrences
- If the user does not specify a variable, the transformation starts an interaction to ask for a variable. It gives a list of variables which can be reached from the selected function clause.
- Side conditions:
  - The new variable name does not exist in the scope of the variable
- Transformations steps and compensations:
  - Replace every occurrence of the variable with the new name. In case of variable shadowing, other variables with the same name are not modified.

2.2. Rename X to Y

```
  max(X,Z) ->
   if
      X=<Z -> Z;
      X>=Z -> X
   end.
```

⇒

```
max(Y,Z) ->
   if
      Y=<Z -> Z;
      Y>=Z -> Y
   end.
```

2.3. reftr_rename_var.erl

```
  -module(reftr_rename_var).
-vsn("$Rev: 10106 $ ").

%% Callbacks
-export([prepare/1]).

-include("user.hrl").

%%% ========================
%%% Callbacks

%% @private
prepare(Args) ->
```

2.4. Querying the Arguments

- The selected variable:

```
Var = ?Args:variable(Args, rename)
```


Created by XMLmind XSL-FO Converter.

- The name of the variable:

```
OldVarName = ?Var:name(Var)
```

2.5. Querying information to check the side conditions

- Occurrences of the variable:

```
Occs = ?Query:exec(Var, ?Var:occurrences())
```

- Visible variables:

```
Visibles = ?Query:exec(Occs,
                   ?Expr:visible_vars())
```

- Used variables:

```
Useds =
?Query:exec(Var,
    ?Query:seq(?Var:scopes(),
               ?Clause:variables()))
```

- Potential name clashes:

```
ClashNames  =
[?Var:name(V) || V <- Visibles ++ Useds]
```

2.6. Asking the new variable name

```
  References  =
    ?Query:exec(Var,?Var:references()),
RefNum      =
    length(References),

CheckParams =
    {OldVarName, ClashNames, RefNum},

ArgsInfo    =
    add_transformation_info(Args, Var, Occs),

NewName     =
    ?Args:ask(ArgsInfo, varname,
              fun cc_newname/2, fun cc_error/3,
              CheckParams),
```

2.7. Performing the transformation

```
  add_transformation_info(Args, Var, Occs) ->
     VarName = ?Var:name(Var),
     OccLen  = length(Occs),
     Info    =
         ?MISC:format("Renaming variable ~p (~p occurrences)",
                      [VarName, OccLen]),
     [{transformation_text, Info} | Args].
```

2.8. Performing the transformation

Created by XMLmind XSL-FO Converter.

```
 [fun () ->
   ?Macro:inline_single_virtuals(Occs, elex),
   [?Macro:update_macro(VarOcc,
                        {elex, 1},
                        NewName)
        || VarOcc <- Occs],
   hd(Occs)
end,
fun(Occ)->
   [hd(?Query:exec(Occ,?Expr:variables())) ]
end].
```

2.9. Side condition checking with interaction

```
 cc_newname(NewVarName, {OldVarName, ClashNames, RefNum}) ->
    ?Check(NewVarName =/= OldVarName,
           ?RefError(new_varname_identical, OldVarName)),
    ?Check((NewVarName =/= "_") or (RefNum == 0),
           ?RefError(underscore_not_allowed, OldVarName)),
    ?Check(not lists:member(NewVarName, ClashNames),
           ?RefError(var_exists, NewVarName)),
    NewVarName.

cc_error(?RefError(Type, Info), NewVarName, _Tuple) ->
    case Type of
        new_varname_identical ->
            ?MISC:format("The variable name is already ~p",
                          [Info]);
        var_exists ->
            ?MISC:format("The variable ~p is already used.",
                         [NewVarName]);
        underscore_not_allowed ->
            ?MISC:format("Variable ~p cannot be replaced" ++
                         " with an underscore (used variable).",
                         [Info])
    end.
```

2.10. Exercise

- Write your own refactoring!

Created by XMLmind XSL-FO Converter.

Chapter 26. Practice 12

1. Duplicated Code Detection

1.1. Code Duplicates

- Source code fragments that occur several times in the program
- Identical or similar code clones
- Result of copy&paste programming
- Bad smells: reduces the quality and maintainability
- Increase the possibility of errors

1.2. Duplicate Code Detectors

- Hard to identify clones on large-scale projects
- Tool support is required
- Different approaches:
  - line based detection
  - token/AST based detection
  - syntax/metric based detection
  - mixed solutions

1.3. Clone IdentifiErl

- Component of RefactorErl
- Two clone detector algorithms
-

```
ri:clone_identifierl([{algorithm, A},
                      {cache, L},
                      {func_list, [{M,F,A}]}]).
```

  - algorithm: `matrix, sw_metrics`
  - cache: `false, true`
  - func_list: list of functions, `[{M,F,A}]`

1.4. Clone IdentifiErl

-

```
ri:clone_identifierl([{algorithm, matrix},
                        {cache, true}])
```


Created by XMLmind XSL-FO Converter.

-

```
ri:clone_identifierl([{algorithm, matrix},
                      {cache, false},
                      {func_list,[{mymod, myfunc, 0},
                                  {dupmod, dupfun, 0}]}])
```

-

```
ri:clone_identifierl([{algorithm, sw_metrics},
                      {cache, true},
                      {func_list,[{mymod, myfunc, 0},
                                  {dupmod, dupfun, 0}]}])
```

- `ri:clone_identifierl()`

1.5. Clone IdentifiErl

```
 ri:clone_identifierl([{algorithm, sw_metrics}]).

...

--------------------------------------------------------------------------------
Left one (found in refusr_cyclic_fun):
check_edge(Graph, V1, V2, Label)->
    case edge(Graph, V1, V2) of
        [] -> digraph:add_edge(Graph, V1, V2, Label);
        _ -> ok
    end.

Right one (found in refusr_cyclic_mod):
check_edge(Graph, Vertex1, Vertex2)->
    case edge(Graph, Vertex1, Vertex2) of
        [] ->
             digraph:add_edge(Graph, Vertex1, Vertex2);
        _ -> ok
    end.
--------------------------------------------------------------------------------
```

1.6. Search Duplicates

- Component of RefactorErl
- Suffix-tree based algorithm
-

```
ri:search_duplicates([{Key, Value}])
```

- Optional parameters:
  - `{files, Files}`- Define the files in which the search is carried out.

    ```
    Files = [ Filepath::string() |
       File::string() |
       RegExp::string() |
       Module::atom() ]
    ```

    There are four options to specify files:

Created by XMLmind XSL-FO Converter.

- path of the file
- regexp and file, that contains regexps
- name of the module

By default all files from the database is analysed.

1.7. Search Duplicates

-
```
ri:search_duplicates([{Key, Value}]).
```

- `{minlen, integer()}` - Define the minimal length of a clone. Default value: 10.
- `{minnum, integer()}` - Define the minimal number of clones in a clone group. Default value: 2.
- `{overlap, integer()}` - Define the scale of the overlap enabled. Default value: 0.
- `{output, Filename::string()}` - Specify the name of the file in witch the results are saved.
- `{name, Name::atom()}` - Specifies the name of the search. Use this name to save the result. Default value refers to the parameters of the search.

1.8. Search Duplicates

- `stored_dupcode_results/0` - queries all saved results (name and information about the parameters of the analysis)
- `save_dupcode_result/2` - saves the given result in the file:
  - `Name::atom()` - name associated with the result
  - `Filename::string()` - name of the file
  -
```
ri:save_dupcode_result(temp20120527205412,
                       "result.txt")
```
- `show_dupcode/1` - lists all clone groups of the result:
  - `Name::atom()` - name associated with the result
  - `ri:show_dupcode(temp20120527205412)`
- `show_dupcode_group/2` - lists the given clone group of the result:
  - `Name::atom()` - name associated with the result
  - `GroupNumber::integer()` - the number of the required clone group
  - `ri:show_dupcode_group(temp20120527205412,2)`

1.9. Search Duplicates

```
ri:search_duplicates([
        {files, [dup2,
```


Created by XMLmind XSL-FO Converter.

```
                 "/home/user/dups/dup1.erl",
                 "/home/[0-9a-zA-Z/_.\-]+/src"]},
        {minlen, 30}]).

Initial clone detection started.
Initial clone detection finished.
Trimming clones started.
Trimming clones finished.
Filter clones finished.
Calculating positions started.
Calculating positions finished.
2 clone groups found.
Result saved to temp20120527234307...
run ri:show_dupcode(temp20120527234307) to see the result.
ok

ri:search_duplicates([{files,[dup1,dup2]}]).

2 clone groups found.
Result saved to temp20120527234307...
run ri:show_dupcode(temp20120527234307) to see the result.
ok
```

1.10. Search Duplicates

```
 ri:search_duplicates([{files,[dup1,dup2]}]).

2 clone groups found.
Result saved to temp20120527234307...
run ri:show_dupcode(temp20120527234307) to see the result.
ok
```

1.11. Search Duplicates

```
 ri:show_dupcode(temp20120527234307).

[{1,
  [[{filepath,"/home/user/dups/dup1.erl"},
    {startpos,{8,1}},
    {endpos,{18,11}}],
   [{filepath,"/home/user/dups/dup2.erl"},
    {startpos,{5,1}},
    {endpos,{15,11}}]]},
 {2,
  [[{filepath,"/home/user/dups/dup1.erl"},
    {startpos,{6,5}},
    {endpos,{6,62}}],
   [{filepath,"/home/user/dups/dup2.erl"},
    {startpos,{18,5}},
    {endpos,{18,58}}]]}]
Contains 2 clone groups...
run ri:show_dupcode_group(temp20120527234307,
                          GroupNumber::integer())
to see one of the clone groups.
ok
```

1.12. Search Duplicates

```
 ri:show_dupcode_group(temp20120527234307,2).

[[{filepath,"/home/user/dups/dup1.erl"},
  {startpos,{6,5}},
  {endpos,{6,62}}],
```


Created by XMLmind XSL-FO Converter.

```
 [{filepath,"/home/user/dups/dup2.erl"},
  {startpos,{18,5}},
  {endpos,{18,58}}]]
Contains 2 members.
ok
```

2. Eliminating Code Clones with Refactorings

2.1. Using Refactorings

- Identify code clones
- Use refactorings to eliminate them:
  - Generalize function with *Introduce function parameter*
  - Create new function with *Introduce function*
  - Optionally: Folding against a function definition (not implemented in RefactorErl)
  - Introduce/eliminate import when it is necessary
  - Do moving or renaming when makes sense

2.2. Eliminating clones

```
 --------------------------------------------------------------------------------
Left one (found in dup):
check_1(P1, P2)->
    case property1(P1, P2) of
        true -> ok;
        _      -> nok
    end.

Right one (found in dup)
check_2(P1, P2)->
    case property2(P1, P2) of
        true -> ok;
        _      -> nok
    end.
--------------------------------------------------------------------------------
```

2.3. Generalize over the function call

```
 -module(dup).

call_props(P1, P2) ->
    check_1(P1, P2),
    check_2(P1, P2).

check_1(P1, P2)->
    case property1(P1, P2) of
        true -> ok;
        _      -> nok
    end.

check_2(P1, P2)->
    case property2(P1, P2) of
        true -> ok;
        _      -> nok
    end.
```


Created by XMLmind XSL-FO Converter.

2.4. Introducing the general check function

```
 -module(dup).

call_props(P1, P2) ->
    check_1(P1, P2, fun(P1, P2) -> property1(P1, P2) end),
    check_2(P1, P2, fun(P1, P2) -> property2(P1, P2) end).

check_1(P1, P2, Fun)->
    case Fun(P1, P2) of
        true -> ok;
        _    -> nok
    end.


check_2(P1, P2, Fun)->
    case Fun(P1, P2) of
        true -> ok;
        _    -> nok
    end.
```

2.5. Change the call in check_2

```
 -module(dup).

call_props(P1, P2) ->
    check_1(P1, P2, fun(P1, P2) -> property1(P1, P2) end),
    check_2(P1, P2, fun(P1, P2) -> property2(P1, P2) end).

check_1(P1, P2, Fun)->
    check(Fun, P1, P2).

check(Fun, P1, P2) ->
    case Fun(P1, P2) of
        true -> ok;
        _ -> nok
    end.

check_2(P1, P2, Fun)->
    case Fun(P1, P2) of
        true -> ok;
        _    -> nok
    end.
```

2.6. Done!

```
 -module(dup).

call_props(P1, P2) ->
    check_1(P1, P2, fun(P1, P2) -> property1(P1, P2) end),
    check_2(P1, P2, fun(P1, P2) -> property2(P1, P2) end).

check_1(P1, P2, Fun)->
    check(Fun, P1, P2).

check(Fun, P1, P2) ->
    case Fun(P1, P2) of
        true -> ok;
        _ -> nok
    end.

check_2(P1, P2, Fun)->
    check(Fun, P1, P2).
```


Created by XMLmind XSL-FO Converter.

2.7. Eliminate this clone!

```
 -------------------------------------------------------------------------------
Left one (found in refusr_cyclic_fun):
check_edge(Graph, V1, V2, Label)->
    case edge(Graph, V1, V2) of
        [] -> digraph:add_edge(Graph, V1, V2, Label);
        _ -> ok
    end.

Right one (found in refusr_cyclic_mod):
check_vertex(Graph, Vertex)->
     case  digraph:vertex(Graph, Vertex) of
         false -> digraph:add_vertex(Graph, Vertex);
         _ -> ok
     end.

-------------------------------------------------------------------------------
```

2.8. Exercise

- Find code clones in the source code of Mnesia!
- Eliminate some clone pairs with RefactorErl!

Created by XMLmind XSL-FO Converter.

Chapter 27. Practice 13

1. Data-flow Analysis Introduction

1.1. Data-flow

- Gathering information about data handling and manipulation
- Possible sets of values at various points
- Different data-flow analyses:
  - constant-propagation
  - liveness analysis
  - available expression analysis
  - reaching definition analysis
  - etc.

1.2. Reaching definition analysis

- Erlang is a single assignment language, thus our interest is in reaching definition analysis
- Find those program points that can be a copy of a certain expression or variable
- The result of the analysis is a Data-Flow Graph (DFG)
- The DFG includes the direct and indirect relations among expressions
- DFG = (N , E)
  - $n_i \in N$ are nodes in the graph
  - $(n_i \rightarrow n_j) \in E$ are edges of the graph

1.3. Kinds of Data-Flow edges

- $n_i \overset{f}{\rightarrow} n_j$ – the node n_j can be a copy of n_i
- $n_i \overset{c_k}{\rightarrow} n_j$ – the node n_j is a compound expression that contains the value of node n_i as its kth element
- $n_i \overset{s_k}{\rightarrow} n_j$ – the node n_j is the kth element of the compound expression n_i
- $n_i \overset{d}{\rightarrow} n_j$ – the node n_j directly depends on the node n_i

1.4. Data-Flow reaching

- Indirect data-flow can be computed from the data-flow graph by calculating the transitive closure of the graph
- The transitive closure is refined with a reaching relation
- Levels of the analysis:

Created by XMLmind XSL-FO Converter.

- 0th order analysis
- 1st order analysis

$\vdots$

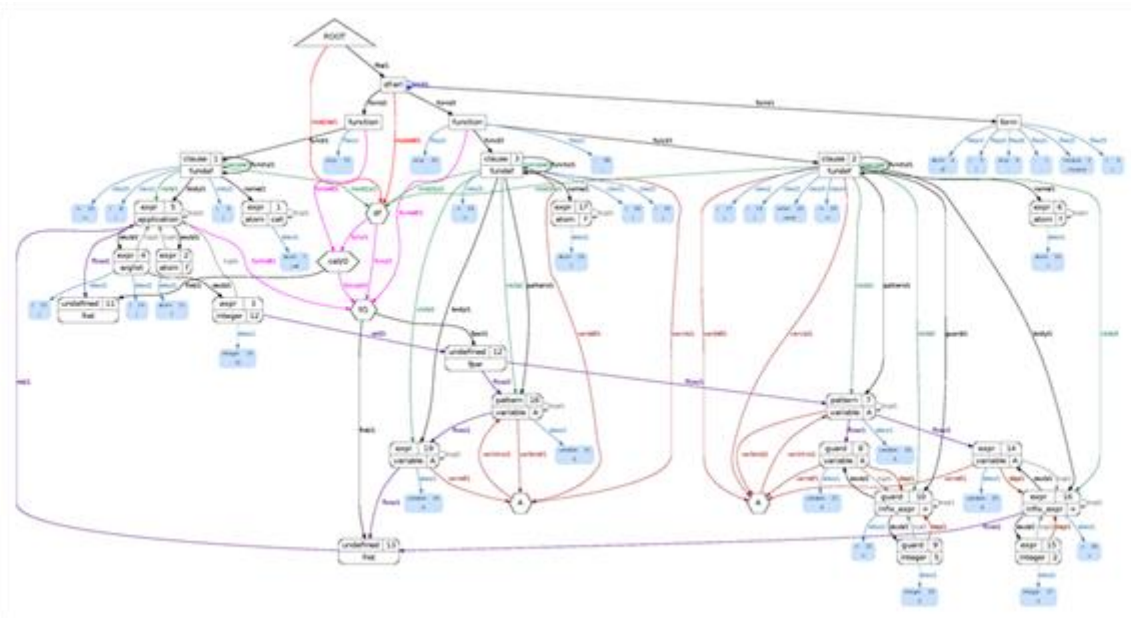
1.5. DFG in RefactorErl

- Built by the data-flow analyser (`refanal_dataflow.erl`)
- Asynchronous
- During initial loading
- Syntax based: inserting, removing or updating a syntax-tree node triggers the analysis
- The Semantic Program Graph contains the data-flow edges

1.6. Example Graph

- `ri:add("df.erl")`
- `ri:svg()`

1.7. Example Graph



1.8. Exercise

- Examine the data-flow analyser!

2. Reaching in RefactorErl

2.1. Semantic Queries

- Reaching is available through the query language
- `@expr.orgin`

Created by XMLmind XSL-FO Converter.

- `@expr.reach`

2.2. Reaching in refanal_dataflow.erl

- `refanal_dataflow:reach(Nodes,Opts,Compact)`
- `Nodes = [node()]` - list of graph nodes
- `Opts = [{Key, Value}]` - list of options:
    - `{back, bool()}` - the direction of the reaching. `true` is for backward, `false` is for forward reaching.

      Default value is `false`: the result set contains expressions that may return the result of one of the source expressions
    - `{safe, bool()}` - `true` means that the expressions in the result set cannot return values independent of the source set. `false` means that the expressions in the result set may return a value that is independent of the source set. This is the default behaviour.
- `Compact = bool()` - `true` returns the compact data-flow reaching, `false` returns the whole result of data-flow reaching.

2.3. Reaching in refanal_dataflow.erl

- First-order data-flow reaching:

```
            refanal_dataflow:reach_1st(Nodes,
                                       Opts,
                                       Compact)
```

- The parameters are the same as in the zeroth order analysis

2.4. Exercise

- Use the data-flow graph to draw the flow of a data from a certain point of the program
- You can extend the implementation of the reaching algorithm

Created by XMLmind XSL-FO Converter.

Chapter 28. Practice 14

1. Building the DB

1.1. Running example

```
  -module(factorial).

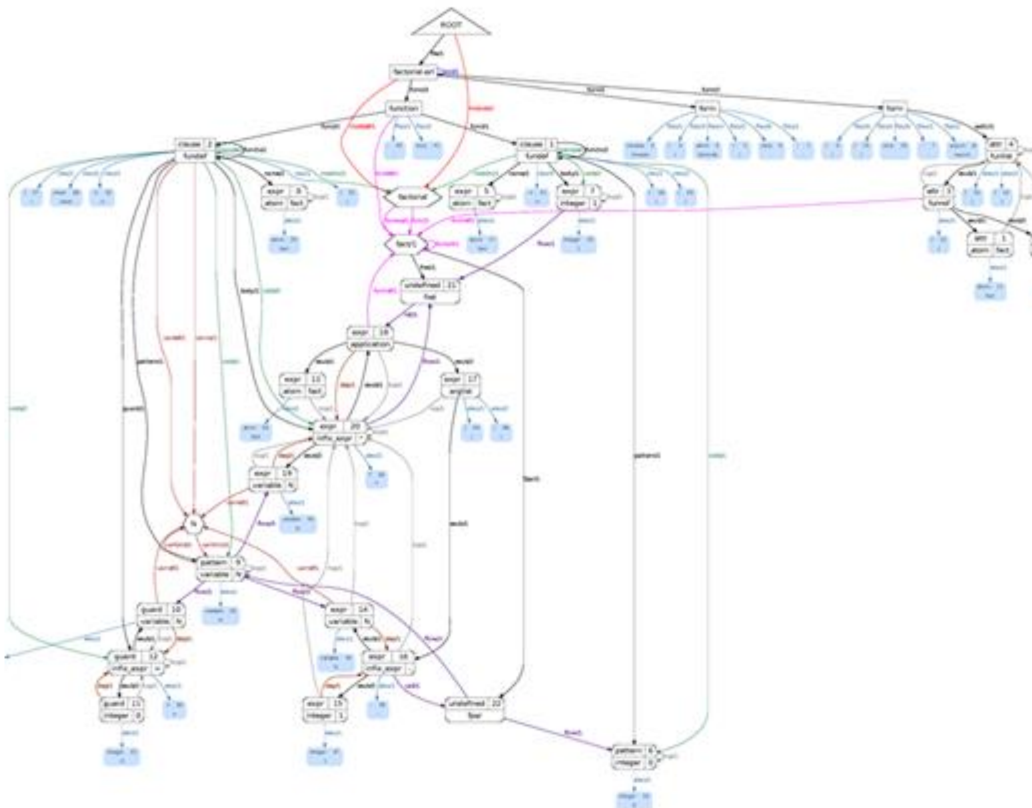
-export([fact/1]).

fact(0) ->
    1;
fact(N) when N > 0 ->
    fact(N-1)*N.
```

1.2. Building the database

- `ri:add("factorial.erl")`
- `ri:svg()`

1.3. SPG of factorial



2. Control-Flow Graph

2.1. Calculating the Control Flow Graph

- Separately calculated for every function

Created by XMLmind XSL-FO Converter.

- Implemented in
  - `refsc_cfg_server.erl` – implemented as a gen_server application
  - `refsc_cfg_utils.erl` – different utilities for calculating the CFGs (algorithm for calculating, helper functions)
- `refsc_cfg_server` exports:
  - Server managing: `start_link/0`, `stop/0`, `reset/0`
  - Asynchronous communication: `build_cfgs/1`, `rebuild_cfgs/1`, `delete_cfgs/1`
  - Synchronous communication: `get_cfg/1`, `get_status/1`

2.2. Calculating the Control Flow Graph – Managing the Server

- `start_link()` : starts the server
- `stop()` : stops the server
- `reset()`: rests the cashed CFG graphs

2.3. Calculating the Control Flow Graph – Asynchronous communication

- `build_cfgs(FormList)`
  - `FormList`: list of SPG forms
  - Builds CFG for functions included in the argument list
  - It builds the CFGs only for functions not cashed in the server (if the CFG was previously built)
- `rebuild_cfgs(FormList)`
  - `FormList`: list of SPG forms
  - Builds CFG for functions included in the argument list
  - Deletes the cashed CFGs and initiates the build process for every function included in the list
- `delete_cfgs(FormList)`
  - `FormList`: list of SPG forms
  - Deletes the cashed CFGs for the given functions (if there is any)

2.4. Calculating the Control Flow Graph – Synchronous communication

- `get_cfg(Form)`
  - `Form`: SPG form ID
  - Asking the server for CFG, it assumes that the build process was previously initiated for the given form
  - Return value: `{Form, Edges, AppNodes}`
    - `Form`: the form same as the argument

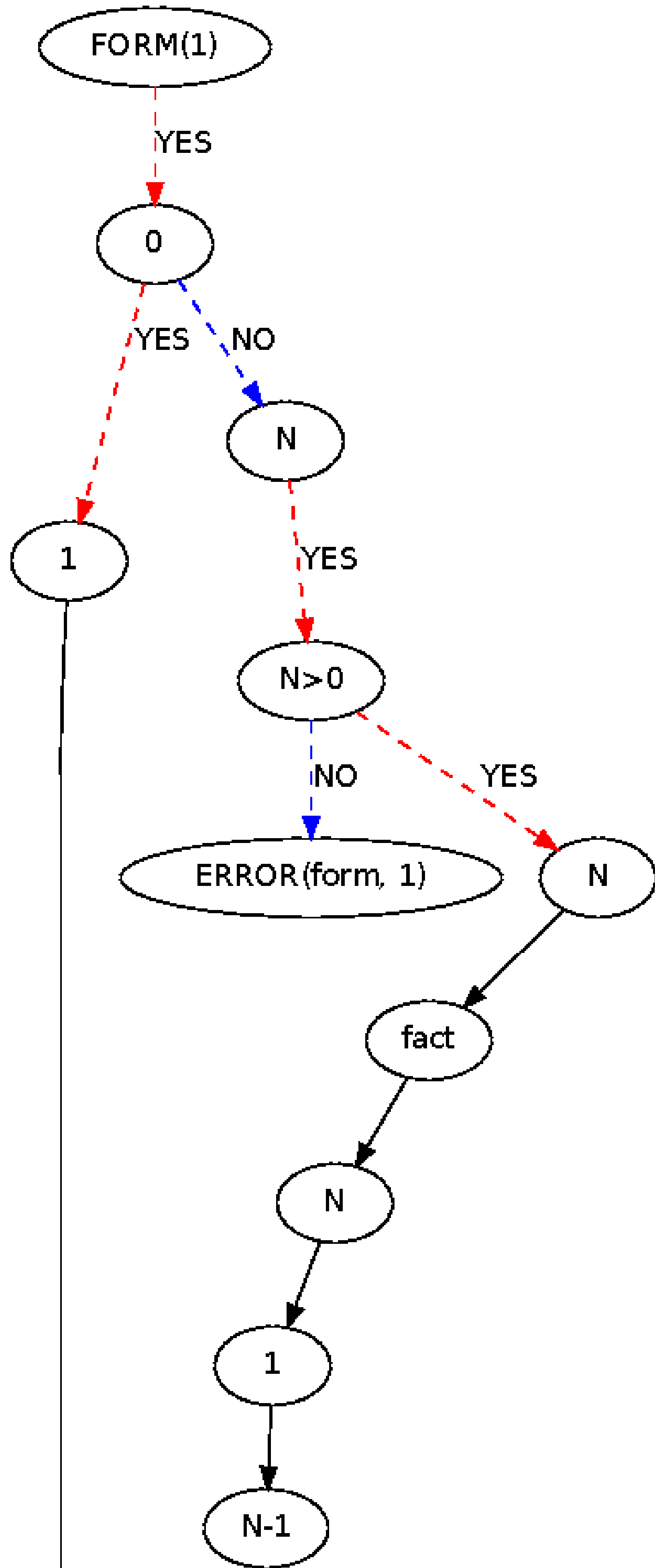
Created by XMLmind XSL-FO Converter.

    - `Edges`: list of edges (`{SNode, ENode, Label}`)
    - `AppNodes`: list of application nodes found in the function

- `get_status(Form)`
  - `Form`: SPG form ID
  - Asking the progress of calculation
  - Return value: `ready | in_progress | unavailable`
    - `ready` : the CFG is ready
    - `in_progress` : the CFG calculation has been started, but not finished yet
    - `unavailable`: the calculation of the CFG is not possible (e.g. erroneous form in the SPG)

2.5. Example CFG

Created by XMLmind XSL-FO Converter.


FORM(1)
YES
0
YES
NO
1
N
YES
N>0
NO
YES
ERROR(form, 1)
N
fact
N
1
N-1

2.6. Exercise

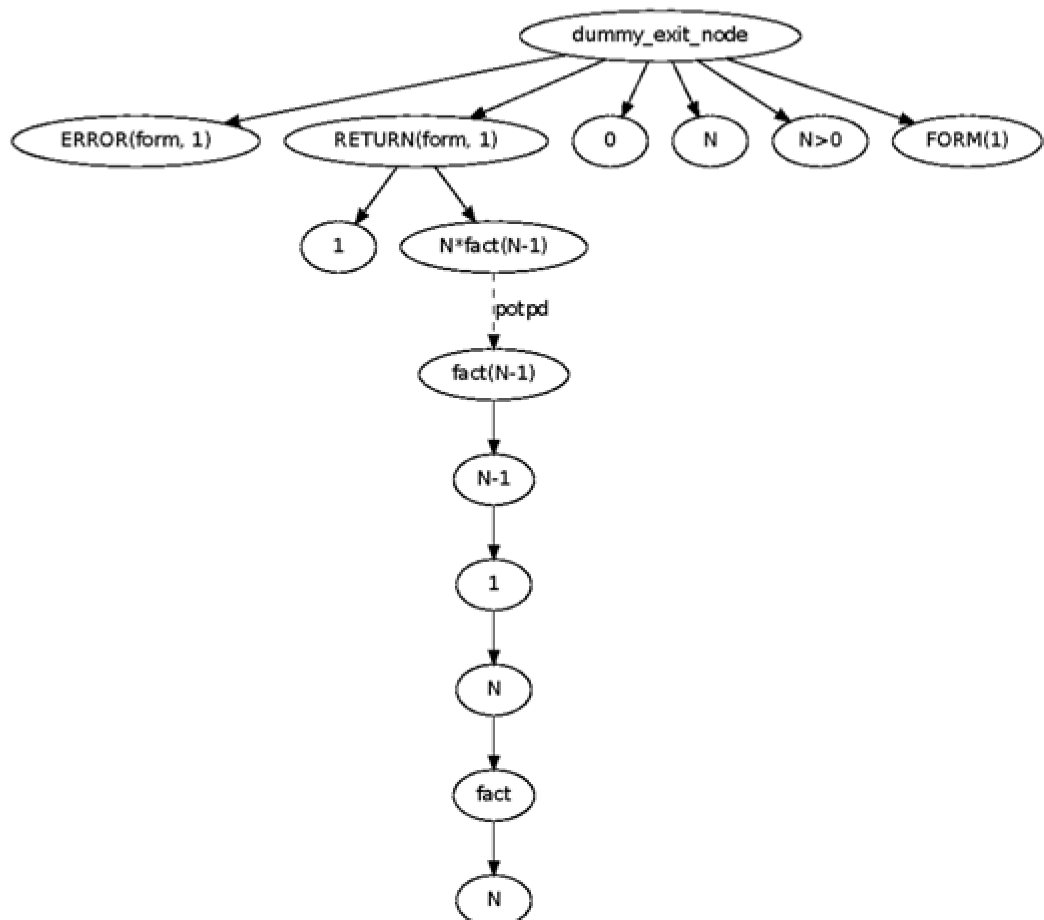
- Check the CFG of factorial!
- Validate it based on the CFG building rules!

3. Postdominator Tree

3.1. Calculating the Postdominator Tree

- Implemented in `refsc_postdom.erl`
- Calculates the Immediate Postdominator Tree from the CFG
- Interface: `immediate_postdominators(Form, CFGEdges, AppNodes)`
  - Return value : `{Edges, Type}`
    - `Edges`: list of edges (`{SNode, ENode, Label}`)
    - `Type`: type is `forest` (if function may fail) and `tree` (function exits normally)

3.2. Example PDT



3.3. Exercise

Created by XMLmind XSL-FO Converter.

- Check the PDT of factorial!
- Validate it based on its CFG!

4. Dependence Graph

4.1. Calculating the Control Dependence Graph

- Separately calculates for every function
- Calculates the compound control dependence graph
- Implemented in
  - `refsc_cdg_server.erl` – implemented as a gen_server application
  - `refsc_cdg_utils.erl` – different utilities for calculating the CDG (algorithm for calculating, helper functions)
- `refsc_cdg_server` exports:
  - Server managing: `start_link/0`, `stop/0`, `reset/1`
  - Asynchronous communication: `build/1`
  - Synchronous communication: `get_compound_cdg/1`, `get/1`

4.2. Calculating the Control Dependence Graph – Managing the Server

- `start_link()` : starts the server
- `stop()` : stops the server
- `reset(Mode)`
  - `cdg_reset`: resets the cashed CDGs
  - `full_reset`: resets the cashed CDGs and triggers the CFG reset

4.3. Calculating the Control Dependence Graph – Asynchronous communication

- `build(FormList)` :
  - `FormList`: list of form identifiers from SPG
  - initiates the CFG and the DFG building process for the given list of forms (rebuilding)

4.4. Calculating the Control Dependence Graph – Synchronous communication

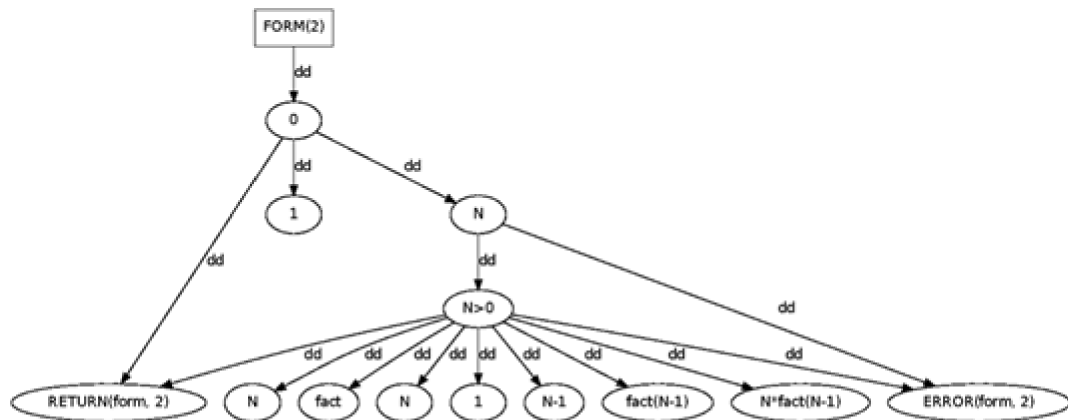
- `get(FormList)` :
  - `FormList`: list of form identifiers from SPG
  - initiates the building process and returns the CDGs for every function in a list
  - Return value: `[Form, PDInfo, Edges, CallSrc]`

Created by XMLmind XSL-FO Converter.

- `Form`: the function form
- `PDInfo`: immediate postdominator tree and the its type
- `Edges`: list of edges (`{SNode, ENode, Label}`)
- `CallSrc`: application nodes for further analysis

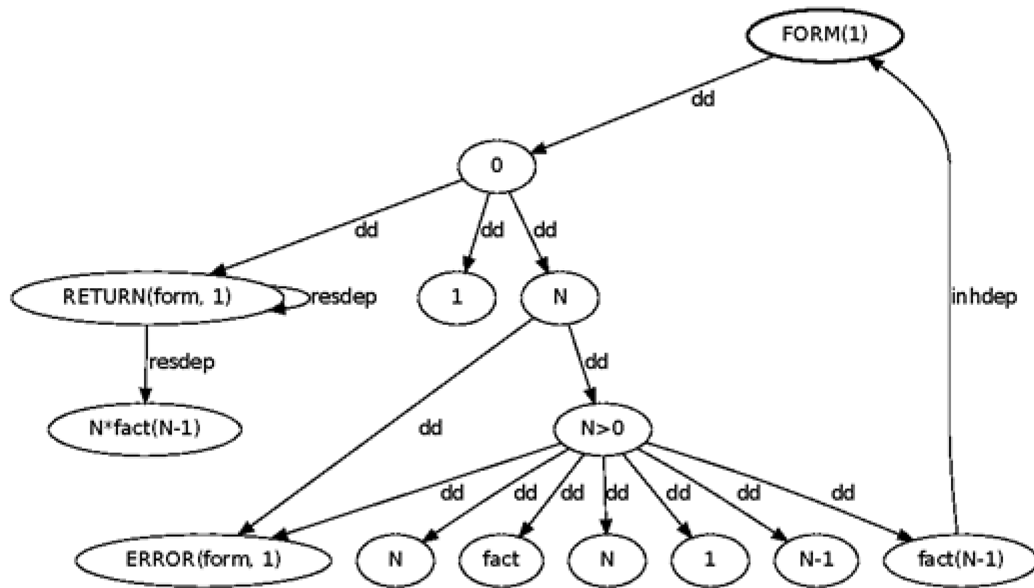
- `get_compound_cdg(FormList)` :
  - `FormList`: list of form identifiers from SPG
  - initiates the CFG and the DFG building process for the given list of forms (rebuilding) and resolves interfunctional dependencies
  - Return value: `[Edge]`
    - `Edge`: tuples representing edges (`SNode, ENode, Label`)

4.5. Example CDG



4.6. Example CCDG

Created by XMLmind XSL-FO Converter.



4.7. Exercise

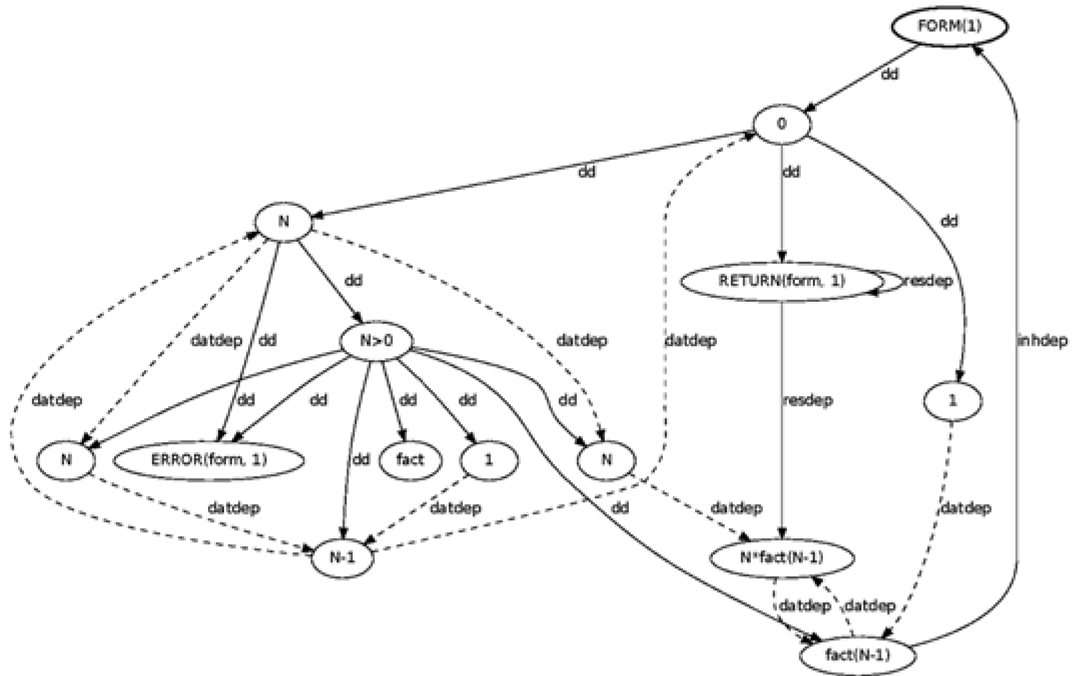
- Check the CDG and CCDG of factorial!
- Validate it based on its CFG and PDT!

4.8. Calculating the Dependence Graph

- Extending the previously generated graphs with data-flow and data dependency
- Results a more accurate graph
- This graph can be used for further analysis

4.9. Example DG

Created by XMLmind XSL-FO Converter.



5. Exercises

5.1. Exercises

- Calculate the graphs for function `fibonacci/1`!
- Use the control-flow graph to draw the control flow from a certain point of the program!
- You can use your data-flow implementation, if it is general!
- Write your own query language to query information from the CFG and DG!

Created by XMLmind XSL-FO Converter.