From the Department of Computer Science

Concurrency correctness in Scala

A Thesis submitted in fulfilment
of the requirements
for the Degree of

Master in Computer Science

at the University of Auckland

by
Marvin Hansen
Std. ID: 2997334

Auckland, New Zealand, 2013
“There is a time for everything, 
and a season for every activity under the heavens.”

*Ecclesiastes, third century BC.*
Abstract

The ongoing effort of transforming multi-core CPUs towards many integrated core APUs (Accelerated Computation Unit) by converging CPUs and GPUs into one single chip, changes the previously linear growth of CPU power towards exponential growth in APU computational power. Addressing the complexity of concurrency programming in the context of changing hardware remains an ongoing challenge. In order to overcome limitations of conventional concurrency programming as part of mainstream programming languages, I have investigated ways to verify the correctness of the concurrency code in Scala.

Initially, I investigated the transformation of software contracts into test-oracles in order to perform fully automated testing, a concept known from ongoing research in Eiffel. In order to do that, I investigated several Design by Contracts tools and finally developed f-DBC, a novel tool based on a flexible component architecture for supporting functional Design by Contract in Scala. An analysis of f-DBC benchmark results provides evidence that contracts written in f-DBC do not cause any measurable reduction in runtime performance compared to the same application without contracts.

However, an empirical analysis of eight Eiffel and twelve Scala projects has provided evidence that contracts are seldom written in real world projects. Based on this insight, I have developed the main contribution of this thesis: a macro for the Scala compiler that replaces the initial approach with compiler checks on pure functions. Even though the concept of function purity is already known, surprisingly little work has been done for verifying function purity. The developed macro introduces a first compiler-based verification of function purity and a few more aspects relevant to concurrency programming. Altogether, the macro reports nine categories of thread-safety issues. Once a function guarded by the macro compiles, it is verified to be thread-safe in any concurrent execution model. A set of examples for each category illustrates the usefulness of the compiler macro. Future work may extend the approach further into other fields such as distributed programming.
Statutory Declaration

I hereby declare that this thesis is the result of my own work and that no other than the indicated aids have been used for its completion. Material borrowed directly or indirectly from the works of others is indicated in each individual case by acknowledgement of the source and also the secondary literature used.

This work has not previously been submitted to any other examining authority and has not yet been published.

After completion of the examining process, this work will be given to the library of the University of Auckland (UoA), where it will be accessible to the public for viewing and borrowing. As author of this work, I agree to this procedure.

Auckland, New Zealand, 11 November 2013
Open Access

All content and software crafted during the time of this master thesis is thereby unrestricted accessible to the public. A thesis website contains a digital copy of this report, all meeting notes, collected data and access to all source code. The website is available online at:

http://marvin-hansen.github.io/Msc/¹

Furthermore, all software is licensed under the MIT licence allowing unrestricted rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software. The full licence text is located in Appendix A or accessible online at: http://mit-license.org/

¹A secondary link is: https://github.com/marvin-hansen/Msc
Acknowledgments

I would like to express my gratitude to Associate Professor Ewan Tempero for giving me the opportunity to develop and write this thesis at the Department of Computer Science at the University of Auckland. Furthermore, I want to thank my family for their support throughout my life and especially during my studies and the development and writing of this thesis. And I thank all my friends and colleagues for all their help, support, interest and valuable suggestions.
## Contents

1 Introduction .................................................. 1

2 Motivation .................................................. 6  
   2.1 Question of research .................................... 12  
   2.2 Aim .................................................. 12

3 Design by Contract ............................................. 13  
   3.1 Overview ............................................... 13  
      3.1.1 Precondition ...................................... 15  
      3.1.2 Invariance ........................................ 16  
      3.1.3 Postcondition ...................................... 16  
      3.1.4 Liskov Substitution Principle .................... 17  
   3.2 DBC in Eiffel ......................................... 18  
      3.2.1 Contract inference ................................ 19  
      3.2.2 DBC and automated testing ....................... 20  
      3.2.3 DBC and automated program fixing ............... 24  
   3.3 DBC in Java ........................................... 26  
      3.3.1 Overview ......................................... 26  
      3.3.2 JML based contract tools ......................... 27  
      3.3.3 Dependency injection based contract tools ....... 28  
      3.3.4 Annotation based contract tools .................. 32  
      3.3.5 Inactive projects ................................ 39  
   3.4 DBC in Scala ........................................... 41  
   3.5 DBC in ADA ............................................. 45  
   3.6 Analysis ............................................... 49
## Scala

4.1 Overview ................................................................. 54
4.2 Scala syntax ............................................................. 56
  4.2.1 Pattern matching .................................................. 57
  4.2.2 Extractor objects .................................................. 60
4.3 Unified object model .................................................. 62
  4.3.1 Collections in Scala ............................................... 63
  4.3.2 Null references .................................................... 64
  4.3.3 Object definitions ............................................... 66
  4.3.4 Companion object ................................................ 66
  4.3.5 Classes ............................................................ 67
  4.3.6 Case classes ...................................................... 68
  4.3.7 Traits ............................................................. 69
4.4 Scala type system ..................................................... 70
  4.4.1 Type inference ..................................................... 70
  4.4.2 Parameterized types .............................................. 74
  4.4.3 Variance annotations ............................................ 74
  4.4.4 Type bounds ...................................................... 76
  4.4.5 Abstract type members ......................................... 77
  4.4.6 Self type annotation ............................................. 79
  4.4.7 Structural typing ................................................ 80
  4.4.8 Existential types ................................................. 83
4.5 Implicits ............................................................... 84
  4.5.1 Implicit conversions ............................................. 84
  4.5.2 Implicit function parameters ................................... 86
  4.5.3 Type views and view bounds ................................... 87
  4.5.4 Generic type constraints ....................................... 90
  4.5.5 Implicit objects ................................................. 94
  4.5.6 Implicit classes ................................................ 96
  4.5.7 Value classes ................................................... 98
  4.5.8 Inlined implicit wrappers ..................................... 101
4.6 Class linearization .................................................. 102
4.7 Modular mixin composition .......................................... 105
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.3.1 Null and mutable check</td>
<td>250</td>
</tr>
<tr>
<td>9.3.2 Unit check</td>
<td>251</td>
</tr>
<tr>
<td>9.3.3 IO checks</td>
<td>252</td>
</tr>
<tr>
<td>9.3.4 Exception check</td>
<td>253</td>
</tr>
<tr>
<td>9.4 Implementation</td>
<td>254</td>
</tr>
<tr>
<td>9.5 Limitations</td>
<td>257</td>
</tr>
<tr>
<td>9.6 Analysis</td>
<td>258</td>
</tr>
<tr>
<td>10 Discussion</td>
<td>259</td>
</tr>
<tr>
<td>11 Future work</td>
<td>263</td>
</tr>
<tr>
<td>11.1 Overview</td>
<td>263</td>
</tr>
<tr>
<td>11.2 Background</td>
<td>265</td>
</tr>
<tr>
<td>11.3 Problem</td>
<td>265</td>
</tr>
<tr>
<td>11.4 State of the art</td>
<td>266</td>
</tr>
<tr>
<td>11.5 Proposal</td>
<td>268</td>
</tr>
<tr>
<td>11.5.1 Implicit object serialization</td>
<td>271</td>
</tr>
<tr>
<td>11.5.2 Implicit bidirectional network transfer</td>
<td>272</td>
</tr>
<tr>
<td>11.5.3 Remote function execution</td>
<td>272</td>
</tr>
<tr>
<td>11.5.4 Elastic fault prevention</td>
<td>274</td>
</tr>
<tr>
<td>11.6 Areas of exploration</td>
<td>276</td>
</tr>
<tr>
<td>12 Conclusion</td>
<td>278</td>
</tr>
<tr>
<td>A Appendix: The MIT License (MIT)</td>
<td>280</td>
</tr>
<tr>
<td>B Appendix: DBC usage script</td>
<td>281</td>
</tr>
<tr>
<td>C Appendix: CBR contract</td>
<td>282</td>
</tr>
<tr>
<td>Bibliography</td>
<td>283</td>
</tr>
</tbody>
</table>
List of Figures

2.1 Separated CPU / GPU hardware architecture [1]................. 7
2.2 hUMA shared CPU / GPU memory and address space [2]........... 9
3.1 Ratio software faults to nr. of parameters [3]..................... 21
3.2 Automated fix process in AutoFix-E................................. 24
3.3 Evidence based fix work-flow in AutoFix-E2 [4]..................... 25
3.4 C4J contract inheritance.............................................. 28
3.5 Scala DBC example..................................................... 42
3.6 Comparison of DBC approaches....................................... 50
4.1 Scala’s unified object model........................................... 62
4.2 Value classes in Scala’s unified object model [5].................... 98
4.3 Service-Oriented Component Model (p. 12 in [6])................... 120
4.4 Side effects in functions ( 1.1. in [7])............................... 130
4.5 Task splitting strategies (p.4 in [8])................................ 151
4.6 Abstraction hierarchy in Scala’s collection [9]....................... 152
4.7 Synchronous and asynchronous actor messages (p. 84 in [10])... 162
7.1 Performance plot of all implementations............................. 220
7.2 Performance SIMD vs. non-SIMD...................................... 221
7.3 Performance SIMD: 4 vs. 16 cores................................... 223
8.1 DBC usage in Eiffel Studio compared to median and average usage . 239
8.2 DBC usage for 12 Scala projects compared to median and average usage241
List of Tables

4.1 Different null references in Scala ............................................. 64

6.1 f-DBC: Comparison to C4S ................................................... 184
6.2 f-DBC core components ....................................................... 189
6.3 f-DBC: Default profiles ....................................................... 207

7.1 f-DBC: Performance Evaluation ............................................... 217
7.2 f-DBC: Performance Evaluation ............................................... 218
7.3 Concurrency performance Evaluation ......................................... 219
7.4 Comparison between disruptor and nested Futures ....................... 224

8.1 LoC for 8 Eiffel projects ..................................................... 236
8.2 LoC for 12 Scala projects ................................................... 237
8.3 DBC usage for 7 Eiffel projects ............................................. 238
8.4 DBC usage in Eiffel Studio .................................................. 239
8.5 DBC usage in 12 Scala projects ............................................. 240
8.6 Precondition ................................................................. 242
8.7 Postcondition ................................................................. 242
List of Definitions

3.1.1 Definition (Liskov substitution principle) ............................................. 17
4.10.1 Definition (Set) .................................................................................... 135
4.10.2 Definition (Cartesian product) .............................................................. 135
4.10.3 Definition (Relation) ............................................................................ 135
4.10.4 Definition (Function) ............................................................................ 136
4.10.5 Definition (Referential transparency) .................................................... 136
4.10.6 Definition (Function purity) ................................................................. 136
4.10.7 Definition (Semigroup) .......................................................................... 137
4.10.8 Definition (Identity element) ................................................................. 137
4.10.9 Definition (Monoid) .............................................................................. 137
4.10.10 Definition (Homomorphisms) ............................................................. 137
4.10.11 Definition (Monad) ............................................................................. 137
4.10.12 Definition (Category) .......................................................................... 138
4.10.13 Definition (Functor) ........................................................................... 138
## Listings

1.1 Macro example in Scala ........................................ 4  
1.2 Macro example compile error ................................. 4  
3.1 DBC in Eiffel .................................................. 18  
3.2 C4J-Team usage example ................................ 29  
3.3 C4J-Team contract ......................................... 29  
3.4 C4J contract .................................................. 30  
3.5 SpringContracts example ................................ 31  
3.6 Chex4j example .............................................. 32  
3.7 Contract4J5 example ....................................... 33  
3.8 Modern Jass example ..................................... 34  
3.9 Cofoja example .............................................. 35  
3.10 oVal example ............................................... 38  
3.11 Interface for Scala DBC example ..................... 43  
3.12 Contract for Scala DBC example ..................... 43  
3.13 Implementation for Scala DBC example .................. 44  
3.14 Implementation for Scala DBC example .................. 44  
3.15 Ada contract example ................................... 46  
3.16 Ada dispatching example ............................... 46  
3.17 Ada type example .......................................... 47  
3.18 Ada type predicate example ............................. 48  
3.19 Ada type invariant example .............................. 49  
4.1 Simple for-loop in Java and Scala ...................... 56  
4.2 Pattern matching in Scala ................................ 58  
4.3 Pattern matching on case classes ...................... 59  
4.4 Extractor in Scala .......................................... 60
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.5</td>
<td>Data encapsulation using extractors</td>
<td>61</td>
</tr>
<tr>
<td>4.6</td>
<td>Scala’s unified collection</td>
<td>63</td>
</tr>
<tr>
<td>4.7</td>
<td>Scala option example</td>
<td>65</td>
</tr>
<tr>
<td>4.8</td>
<td>Scala companion object</td>
<td>67</td>
</tr>
<tr>
<td>4.9</td>
<td>Scala constructor overloading</td>
<td>68</td>
</tr>
<tr>
<td>4.10</td>
<td>Scala case class</td>
<td>69</td>
</tr>
<tr>
<td>4.11</td>
<td>Type inference in Scala</td>
<td>71</td>
</tr>
<tr>
<td>4.12</td>
<td>Incorrect type inference on lambda expressions</td>
<td>72</td>
</tr>
<tr>
<td>4.13</td>
<td>Correct type inference on lambda expressions</td>
<td>73</td>
</tr>
<tr>
<td>4.14</td>
<td>Type parameter in Scala</td>
<td>74</td>
</tr>
<tr>
<td>4.15</td>
<td>Incorrect type variance</td>
<td>75</td>
</tr>
<tr>
<td>4.16</td>
<td>Correct type variance</td>
<td>75</td>
</tr>
<tr>
<td>4.17</td>
<td>Upper type bound in Scala</td>
<td>76</td>
</tr>
<tr>
<td>4.18</td>
<td>Lower type bound in Scala</td>
<td>77</td>
</tr>
<tr>
<td>4.19</td>
<td>Abstract type members in Scala</td>
<td>78</td>
</tr>
<tr>
<td>4.20</td>
<td>Usage of abstract type members</td>
<td>78</td>
</tr>
<tr>
<td>4.21</td>
<td>Self-type reference in Scala</td>
<td>79</td>
</tr>
<tr>
<td>4.22</td>
<td>Structural types in Scala</td>
<td>81</td>
</tr>
<tr>
<td>4.23</td>
<td>Structural self-type reference in Scala</td>
<td>82</td>
</tr>
<tr>
<td>4.24</td>
<td>Implicit example in Scala</td>
<td>84</td>
</tr>
<tr>
<td>4.25</td>
<td>Implicit parameters in Scala</td>
<td>86</td>
</tr>
<tr>
<td>4.26</td>
<td>Type views in Scala</td>
<td>88</td>
</tr>
<tr>
<td>4.27</td>
<td>Usage of type views in Scala</td>
<td>89</td>
</tr>
<tr>
<td>4.28</td>
<td>Type without generic type constrains</td>
<td>92</td>
</tr>
<tr>
<td>4.29</td>
<td>Type with generic type constrains</td>
<td>93</td>
</tr>
<tr>
<td>4.30</td>
<td>Implicit object in Scala</td>
<td>94</td>
</tr>
<tr>
<td>4.31</td>
<td>Implicit objects as implicit parameter</td>
<td>95</td>
</tr>
<tr>
<td>4.32</td>
<td>Implicit class in Scala</td>
<td>96</td>
</tr>
<tr>
<td>4.33</td>
<td>Implicit class used for custom language extension in Scala</td>
<td>97</td>
</tr>
<tr>
<td>4.34</td>
<td>Value class in Scala</td>
<td>98</td>
</tr>
<tr>
<td>4.35</td>
<td>Cases of instantiation of a value class</td>
<td>100</td>
</tr>
<tr>
<td>4.36</td>
<td>Implicit wrapper in Scala</td>
<td>101</td>
</tr>
<tr>
<td>4.37</td>
<td>Class linarization in Scala</td>
<td>104</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>4.38 Mixin composition in Scala</td>
<td>106</td>
<td></td>
</tr>
<tr>
<td>4.39 Extending mixin composition in Scala</td>
<td>107</td>
<td></td>
</tr>
<tr>
<td>4.40 Structural typing in Scala</td>
<td>108</td>
<td></td>
</tr>
<tr>
<td>4.41 Components for DI in Java</td>
<td>114</td>
<td></td>
</tr>
<tr>
<td>4.42 DI using Pico in Java</td>
<td>115</td>
<td></td>
</tr>
<tr>
<td>4.43 First component for cake pattern in Scala</td>
<td>122</td>
<td></td>
</tr>
<tr>
<td>4.44 Second component for cake pattern in Scala</td>
<td>123</td>
<td></td>
</tr>
<tr>
<td>4.45 Cake component composition</td>
<td>124</td>
<td></td>
</tr>
<tr>
<td>4.46 Cake component usage</td>
<td>125</td>
<td></td>
</tr>
<tr>
<td>4.47 Cake component composition</td>
<td>126</td>
<td></td>
</tr>
<tr>
<td>4.48 Dynamic context switching using cake pattern components</td>
<td>127</td>
<td></td>
</tr>
<tr>
<td>4.49 Dependency injection using MacWire</td>
<td>128</td>
<td></td>
</tr>
<tr>
<td>4.50 Simple function in Scala</td>
<td>131</td>
<td></td>
</tr>
<tr>
<td>4.51 Higher ordered function in Scala</td>
<td>132</td>
<td></td>
</tr>
<tr>
<td>4.52 Anonymous function in Scala</td>
<td>133</td>
<td></td>
</tr>
<tr>
<td>4.53 Higher kinded typesin Scala</td>
<td>134</td>
<td></td>
</tr>
<tr>
<td>4.54 Monoid interface in Scala</td>
<td>139</td>
<td></td>
</tr>
<tr>
<td>4.55 Monoid implementation in Scala</td>
<td>140</td>
<td></td>
</tr>
<tr>
<td>4.56 Monoid case classes in Scala</td>
<td>141</td>
<td></td>
</tr>
<tr>
<td>4.57 Functor interface in Scala</td>
<td>142</td>
<td></td>
</tr>
<tr>
<td>4.58 Monad interface in Scala</td>
<td>143</td>
<td></td>
</tr>
<tr>
<td>4.59 Transaction monad in Scala</td>
<td>144</td>
<td></td>
</tr>
<tr>
<td>4.60 Monadic combination Scala</td>
<td>145</td>
<td></td>
</tr>
<tr>
<td>4.61 Monadic combination Scala</td>
<td>145</td>
<td></td>
</tr>
<tr>
<td>4.62 Parallel collection Scala</td>
<td>155</td>
<td></td>
</tr>
<tr>
<td>4.63 Future and promise in Scala</td>
<td>156</td>
<td></td>
</tr>
<tr>
<td>4.64 Future and promise in Scala</td>
<td>157</td>
<td></td>
</tr>
<tr>
<td>4.65 For-comprehension</td>
<td>158</td>
<td></td>
</tr>
<tr>
<td>4.66 Compiler rewrite</td>
<td>158</td>
<td></td>
</tr>
<tr>
<td>4.67 Message types for actors</td>
<td>161</td>
<td></td>
</tr>
<tr>
<td>4.68 Fire and forget message</td>
<td>163</td>
<td></td>
</tr>
<tr>
<td>4.69 Send and receive</td>
<td>163</td>
<td></td>
</tr>
<tr>
<td>4.70 Server actor in Scala</td>
<td>164</td>
<td></td>
</tr>
</tbody>
</table>
4.71 Client actor in Scala ........................................ 165
4.72 Router for actor system ..................................... 167
4.73 Basic actor system in Scala ................................. 169
4.74 Client actor system in Scala ............................... 170
4.75 Disruptor event in Java .................................... 174
4.76 Disruptor ring buffer in Java .............................. 175
6.1 f-DBC aliases .................................................. 182
6.2 f-DBC: Writing contracts ................................... 183
6.3 f-DBC Handler access options ............................. 189
6.4 f-DBC Checker interface .................................... 191
6.5 f-DBC Checker implementation ............................ 192
6.6 f-DBC Handler interface .................................... 192
6.7 f-DBC Exit handler implementation ...................... 193
6.8 f-DBC email handler implementation .................... 194
6.9 f-DBC logger handler implementation ................... 196
6.10 f-DBC Logger interface .................................... 197
6.11 f-DBC SLF4J logger implementation ....................... 199
6.12 f-DBC Composition interface .............................. 201
6.13 f-DBC checked handler composition ...................... 202
6.14 f-DBC logged precondition .................................. 204
6.15 f-DBC: logged invariant check ............................. 205
6.16 f-DBC: logged invariant check ............................. 206
6.17 f-DBC: Mock composition .................................. 206
6.18 f-DBC Dev. profile registry ............................... 209
6.19 f-DBC Dev. profile implementation ....................... 209
6.20 f-DBC Dev. profile and companion object ............... 209
6.21 f-DBC protected profiles ................................... 210
6.22 f-DBC aliases .................................................. 211
7.1 f-DBC contract ................................................ 215
7.2 CBR travel case class ........................................ 230
7.3 Local similarity score ........................................ 231
7.4 Global similarity score ....................................... 232
7.5 SIMD global similarity score ............................... 234
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.1</td>
<td>macro: pure functions</td>
<td>246</td>
</tr>
<tr>
<td>9.2</td>
<td>macro: usage example</td>
<td>247</td>
</tr>
<tr>
<td>9.3</td>
<td>macro: compiler warning</td>
<td>247</td>
</tr>
<tr>
<td>9.4</td>
<td>macro: usage example</td>
<td>248</td>
</tr>
<tr>
<td>9.5</td>
<td>macro: var check</td>
<td>250</td>
</tr>
<tr>
<td>9.6</td>
<td>macro: null check</td>
<td>250</td>
</tr>
<tr>
<td>9.7</td>
<td>macro: reported errors</td>
<td>251</td>
</tr>
<tr>
<td>9.8</td>
<td>macro: UnitChecks</td>
<td>252</td>
</tr>
<tr>
<td>9.9</td>
<td>macro: Input checks</td>
<td>252</td>
</tr>
<tr>
<td>9.10</td>
<td>macro: Output checks</td>
<td>252</td>
</tr>
<tr>
<td>9.11</td>
<td>macro: Exception checks</td>
<td>253</td>
</tr>
<tr>
<td>9.12</td>
<td>macro: keyword definition</td>
<td>254</td>
</tr>
<tr>
<td>9.13</td>
<td>macro: null check implementation</td>
<td>255</td>
</tr>
<tr>
<td>9.14</td>
<td>macro: exception check implementation</td>
<td>256</td>
</tr>
<tr>
<td>11.1</td>
<td>DFC proposed syntax</td>
<td>270</td>
</tr>
<tr>
<td>11.2</td>
<td>Scala pickling example</td>
<td>271</td>
</tr>
<tr>
<td>11.3</td>
<td>Scala code runner</td>
<td>272</td>
</tr>
<tr>
<td>11.4</td>
<td>Scala code runner usage example</td>
<td>273</td>
</tr>
<tr>
<td>B.1</td>
<td>Script for counting DBC usage in Scala</td>
<td>281</td>
</tr>
<tr>
<td>B.2</td>
<td>Script for counting DBC usage in Eiffel</td>
<td>281</td>
</tr>
<tr>
<td>C.1</td>
<td>f-DBC CBR contract</td>
<td>282</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Concurrency programming is widely acknowledged as complex, error prone and difficult to debug. Over the past fifteen years, the semiconductor industry has moved to multi-core architecture in order to accelerate computational speed, which makes concurrency programming an important pillar in modern software development. The implied complexity, however, is one of the major obstacles in exploiting the potential that modern multi-core CPUs offer.

Addressing the complexity issue in the context of changing hardware, I have investigated ways to support the development of fast, efficient and correct concurrency code. My primary focus is on ease of programming while verifying the correctness of concurrent code.

For the scope of this thesis, I have chosen to use Scala as the main programming language as it already offers a higher level of concurrency support through parallel collections, event based actors and composable futures. Despite all progress in simplifying the technique of functional concurrency programming, limited progress has been made in the area of verifying the correctness of concurrency code.

The main topic of this thesis is how to ensure the correctness of concurrency code in Scala. Initially, I investigated Design by Contract (DBC); in particular, the adoption of software contracts as test-oracles to enable fully automated testing. Therefore, I have identified and reviewed fifteen DBC projects for Java and Scals built-in support for software contracts.
The review made it clear that none of the existing tools is powerful enough to serve the purpose of using contracts for generating oracles as input for automated testing. Based on the results of the review, I have formulated missing requirements for automated testing, particularly within the context of functional programming in Scala. Next, I have implemented these requirements in f-DBC; this is an object-functional Design by Contract tool that is flexible to use, yet powerful in enforcing contracts either during compilation, at runtime or both.

A performance evaluation of f-DBC has exposed an average performance loss of under 0.1% percent compared to using no contract checking. The initial idea was to extract post-conditions from contracts and use them as test-oracles in order to determine if a generated test has passed or failed. The only assumption was the presence of contracts. However, before developing further tools, I asked the question: Are contracts actually written in practice?

In order to answer this question, I did an empirical data analysis of DBC usage in nine Eiffel projects having a combined five million lines of code. As it turned out, apart from Eiffel studio IDE, contracts are barely written at all. Invariance checks, often used to guard concurrency in Eiffel, are the least used of all DBC checks. As Scala provides only basic DBC support compared to Eiffel, the next question was: Do Scala developers write contracts?

I did another empirical evaluation on twelve Scala projects with the finding that Scala developers write even fewer contracts than Eiffel developers. In particular, Scala software has twelve times fewer preconditions compared to Eiffel. Unlike Eiffel, post-conditions are not written at all in Scala. Detailed results and methodology of both empirical studies are presented in chapter 8.

Based on the study results, I have drawn the conclusion that DBC by itself is not a practical solution to generate test-oracles required for automated testing. Furthermore, I have learned that instead of checking potentially unsafe code at runtime, as DBC does, a much stricter approach is required, particularly for enforcing thread-safety for concurrency programming.

From this conclusion, I asked the question: What exactly causes software faults in concurrency programming? Essentially, the same question was asked a long time ago and has lead to the concept of pure functions in functional programming. A pure function is a function that does not have any side-effect such as mutable fields.
Chapter 1. Introduction

As most of Scalas concurrency concepts are based on functional concurrency, function purity is a well-known and fully-understood concept to guarantee thread-safety. Surprisingly, there is no direct compiler support to enforce function purity during development.

Based on this insight, I have developed scalarius, a macro that prevents all known side-effects in a function from compilation. Furthermore, the macro prevents all null deferences in Scala from compilation in order to prevent NullPointerExceptions at runtime. The direct result is, once the macro is applied to a function, any potential issue that affects thread-safety issues a compiler error reporting the exact cause and location of the issue.

Consequently, once a function guarded by the macro compiles, it is proven to be thread-safe and can be executed in any concurrency model. As the Scala compiler performs a complete check on the entire abstract syntax tree of a function, there is no loophole left to break thread-safety after compilation. Essentially this macro replaces the initial approach of automated testing with a compile check that entirely prevents non-thread-safe code from compilation.

To illustrate the significance of this contribution further, listing 1.1 is taken from the book *Java Concurrency In Practice* and shows an example of a concurrent cache implementing the `Computable` interface. The example is rewritten in equivalent Scala code because the compiler macro currently only supports Scala.

The example cache is assumed to be thread-safe because it uses a `ConcurrentHashMap` as the main data structure which is known to be thread-safe. Even though this code does compile, it has issues and contains a race-condition. In the worst case, the race-condition (1) leads to cache inconsistency that means, for a correct key, the wrong value will be returned because a second thread has overridden the previous result. Cache inconsistency is a well-known source of oblivious data corruption but is very difficult to isolate once deployed in a production system.

The main difficulty is simply that, the race condition cannot reliably reproduced, which means the code in the example will work as expected for but only a limited time. The race condition is undetectable by using a formal analysis because it produces neither a dead nor a live lock. Instead, it randomly overrides data every once in a while. These kinds of problems were previously unpreventable.

---

1Listing 5.17 on page 105 in [11]
Chapter 1. Introduction

Listing 1.1: Macro example in Scala

Applying the developed macro, by placing the `pure` keyword (1) to the compute function, triggers three compile errors, as shown in listing 1.2. First, the mutable variable for storing the result is reported. Mutable fields are not thread-safe unless guarded by complex locking and synchronization. In nearly every case, they can be replaced either by an immutable field or a returning value from a direct method call. Second, the null value is reported because in certain circumstances, null values are causing null pointer exceptions.

Listing 1.2: Macro example compile error
Chapter 1. Introduction

Third, the *InterruptedException* is reported because exceptions are generally unwanted in any pure function. Essentially, the issue is related to the implied IO operation since an exception needs to be printed to a console which is considered as impure. Also, if the code would be thread-safe, there would be no need to handle such an exception.

These issues are just three examples out of nine categories of issues detected and prevented by the compiler macro. I need to emphasize that, the main contribution of this thesis is an investigation into replacing previously difficult to debug concurrency code with simpler functional concurrency which is additionally fully guarded by a compiler macro preventing any possible wrong-usage that could violate thread-safety. The significance is that, functional concurrency in Scala, unlike traditional concurrency in Java can be verified to be correct as shown by the presented macro.

To the best of my knowledge, such a compile-time concurrency verification is not available in any other mainstream programming language.

As the presented results are based on sophisticated implementation details, the background chapter 4 about Scala is necessary detailed and verbose, laying the foundation for the all results briefly summarized in Chapter 5. Chapter 6 contains details about f-DBC and Chapter 7 contains a case study about the performance impact of f-DBC on different concurrency frameworks. Chapter 8 presents the empirical study of contract usage in Eiffel and Scala which lays the foundation for presenting the macro in Chapter 9. All results are discussed in Chapter 10.

Not stopping at concurrency, I have extended the main contribution to a new concept of distributed functional concurrency programming which is subject to future work. Instead of programming low-level distribution or error-handling details, compiler transformation, implicit conversion and automated fault detection are combined towards distributed functional concurrency. The proposal aims to program hundreds of computers with just five lines of code in under five minutes in a safe and verifiable way. The proposal is based on technical feasibility, already existing technology and evidence gathered during this thesis. Chapter 11 presents the proposal. Chapter 12 finally concludes this master thesis.
Chapter 2

Motivation

Over the past ten years, the semiconductor industry has made a strong move towards multi-core architecture in order to further accelerate computation speed. With the rise of GPU computation on video cards, even more acceleration is based on massive parallelism; for example, a 2002 Pentium 4 has a computational power of 12.24 Giga flops per second (GF/s) whereas a 2012 Core i7-3970X processor achieves 336 GF/s which is equivalent to a 2600% increase in CPU computational power within ten years [12]. Most of this increase is based on executing more instructions in parallel within a CPU.

Putting those numbers in context, a closer look at GPU performance is of particular interest. GPU computational power\(^1\) rose over the same time from 1.2 GF/s\(^2\) to 4301 GF/s\(^3\) in 2012 which is an increase of 13700% in just ten years [12]. Furthermore, the development in GPU computational power rises further at a very high rate. For instance, a mid 2013 GPU\(^4\) already exceeds 7 Teraflops per second which is an additional 180% increase in computational power within 10 months.

---

\(^{1}\)All GPU performance is measured as 32Bit Floating-point operations per second

\(^{2}\)A 2002’s Radeon 9700 Pro

\(^{3}\)A 2012 Radeon HD 7970 GHz Edition

Chapter 2. Motivation

Until recently, GPU programming was complex, error prone and time consuming which has left GPU’s widely unused outside gaming despite the enormous computational power. The commonly known limitations in GPU programming\(^5\) are:

- No official support in high level programming languages such as Java
- Must program GPU portions in a c-like language (CUDA / OpenCL)
- Parallel Programming is still the issue

On the hardware side, the major limitations in GPU programming are based on complex memory management caused by copying data permanently between CPU and GPU through a low performance PCI Express (PCI-E) bus\(^6\). Figure 2.1 illustrates the conventional hardware architecture that connects the CPU with GPU over PCI-E [1].

---

\(^5\)http://www.heatonresearch.com/content/initial-impressions-gpu-programming

Putting the slow CPU to GPU connection into perspective, in 2012 the PCI-E bus in version 3 has a transfer rate of 15.75 GB/s but a consumer grade GForce 650 GTX requires 80 GB/s bandwidth\(^7\). This means that the overall computational speedup is limited by the amount of data transferable over PCI Express. Furthermore, the gap between PCI-E throughput and the required bandwidth is growing rapidly. For instance in early 2013, a GForce 690 would already require 384 GB/s\(^8\) of system bandwidth to match the GPU memory speed. From this perspective, the significant slower PCI-E bus is already a serious problem.

Also, as figure 2.1 shows, memory is fragmented with many caches in different places which is the main reason for complex memory management and complex cache coherence. Additionally, the complexity in programming CPU/GPU systems is rooted in separated memory address spaces and these needs to be mapped between the host (CPU/RAM) and the video device (GPU) that uses an entirely different memory layout.

Approaching the problem of slow interconnection and complex memory, the ongoing development in semiconductor industries converge CPU’s and GPU’s into a single chip in an APU (Accelerated Computation Unit).

Consequently, the trend from the previously linear growth of CPU power now be towards exponential growth [13] in APU computational power, despite the early stage of product availability [12]. The reasons for the exponential increase of computational power of APU’s are three-fold.

First, the current generation of APU’s made a major architectural change towards a high performance heterogeneous unified memory access (hUMA) which means all CPU/GPU components not only share the same memory layout but also share the same address space\(^9\). The consequence of hUMA is a significant reduction in programming effort since data between CPU and GPU is shared by passing memory pointers around instead of copying data over the slow PIC-E Bus. Figure 2.2 illustrates the general concept of heterogeneous unified memory for sharing data between CPU and GPU [2][13][12].

\(^7\)http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-650/specifications
\(^8\)http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-690/specifications
\(^9\)http://hsafoundation.com/standards/
Chapter 2. Motivation

Second, APU’s incorporate state of the art GPU design which is entirely based on massive parallelism. Each advancement in manufacturing smaller circuits increases parallel performance because smaller circuits lead to more cores on the same chip size.

Third, unlike CPU design, APU design is at a very early stage which means there is further optimization potential to explore. In particular, circuits shared between CPU and GPU cores need to be identified, consolidated and efficiently re-designed [2][13].

![Figure 2.2: hUMA shared CPU / GPU memory and address space [2].](image)

The APU design may differ from vendor to vendor but the general concept of a converged CPU/GPU has already been adopted and major semiconductor manufacturing companies such as AMD and Intel have been shipping products¹⁰ based on hUMA from 2013. Programming hUMA uses the established OpenCL standard which is also fully supported by all major semiconductor companies¹¹ such as Intel, AMD, ARM, Nvidia, and Samsung. Furthermore, OpenCL bindings to all major programming platforms such as C++¹², .Net¹³ and Java¹⁴ already exist. Additionally, project Sumatra¹⁵ aims to support APU / GPU acceleration of the JVM.

¹⁰All Intel "Haswell" and AMD A Series models
¹¹[http://www.khronos.org/opencl/](http://www.khronos.org/opencl/)
¹²[http://www.khronos.org/registry/cl/api/1.1/cl.hpp](http://www.khronos.org/registry/cl/api/1.1/cl.hpp)
¹³[http://www.nuget.org/packages/OpenCL.Net](http://www.nuget.org/packages/OpenCL.Net)
¹⁴[https://code.google.com/p/aparapi/](https://code.google.com/p/aparapi/)
¹⁵[http://openjdk.java.net/projects/sumatra/](http://openjdk.java.net/projects/sumatra/)
Considering this major shift in hardware, the current programming model of using concurrency only on selected use-cases is not appropriate any more since the current generation of multi-core CPUs is already barely utilized with executing linear applications. Instead, a ”concurrent by default” programming model is required to fully utilize the potential of modern many-core microprocessors [14].

The challenges for software development in the heterogeneous multi-core age are two-fold. The first challenge is efficient thread handling in order to avoid thread congestion and dead-locks. The challenge of efficient thread handling has been under active research for the past fifteen years and has led to new insights in terms of parallel date structures [15], new scheduling strategies [16] and high performance, low latency inter-thread communication [17].

The second challenge is fast, efficient and correct concurrency programming. Efficiency and correctness are important because low level thread management is not sufficient any more to program magnitudes more cores. Thread locking and shared mutable state may work in a dual or quad-core CPU but an early generation low end APU\textsuperscript{16} already has 128 additional GPU cores which means manual thread programming is not an option any more.

Considering the fact that in 2013, a custom high-end APU made for Sony provides 8 CPU cores and 1152 GPU processing units interconnected using hUMA at 176GB/s\textsuperscript{17}, it all comes down to how fast modern APU’s can be programmed. The most difficult and time consuming part of concurrency programming is debugging, because threads execution varies over time, which means no particular order is guaranteed [11].

\textsuperscript{16}AMD A4-5000 APU
\textsuperscript{17}http://www.psu.com/a019527/
Chapter 2. Motivation

2.1 Question of research

Based on the motivation of scalable concurrency, the question of research is: How can concurrency code verified to be correct?

2.2 Aim

The first goal is investigating design by contract as foundation for fully automated testing of concurrent code.

The second goal the development of required tool(s) for verifying the correctness of concurrently executed code. In particular, investigating type-checking and compiler support in Scala is of interest because of the recent inclusion of an API that allows programming the Scala compiler. The expected result is a prototype that demonstrate the general concept of the approach.

The third goal is an evaluation of the chosen approach on an adequate sample in order to gather quantitative evidence about its usefulness.

The primary goal for this thesis is fast and efficient construction of concurrent applications that are verified to be correct. My primary focus is minimizing debugging time while maximizing efficiency in concurrency programming. Strict verification builds the foundation for scaling applications reliable in both directions: horizontal, with the numbers of processing units (CPU/GPU); and vertical, with the number of nodes in a cluster while still exploiting concurrency on each node.
Chapter 3

Design by Contract

3.1 Overview

Design by Contract (DBC) aims to improve software reliability by adding a contract to a software artifact in order to specify what it requires and, what it is ensuring to preserve, as well as defining the expected result. The first popular DBC implementation in Eiffel is described in section 3.2. From there, section 3.3 takes a closer look at how Design by Contract is works in Java. Next, section 3.4 summarizes Scala’s built-in DBC functionality. Section 3.5 illustrates the contract driven development model in Ada.

Design by Contract was originally intended to provide definite assertions in programming to ensure the correctness of an application. This idea goes back to Alan Turing who, in 1949, formulated a need for formal assertion checking. In 1969, Hoare introduced Hoare triples as formalization of assertion checking where each triple consists of a pre and a postcondition. A check for class invariant was then introduced in 1972 by Hoare as an addition to triples [18][19].

From the 1970s onwards, specifying software more formally becomes part of programming languages such as Gypsy, Alphard and Euclid. Gypsy was the first programming language having built-in capabilities for writing a formal description of software, and can be considered the first approach towards software specifications.
Chapter 3. Design by Contract

Gypsy includes a verification condition generator, algebraic simplifier and interactive theorem prover, allowing program verification at compile as well as during run-time [20][21][22][23].

The Alphard programming language, also developed in the 70’s, supports pre, postcondition, invariant and an initial clause to specify an initial value of a field. Euclid was designed as an addition to the Pascal programming language. It incorporates assertions such as pre, postcondition and invariant written as comments into Pascal programs. These assertions can be ignored, used by a verifier to check the correctness of a program or compiled into run-time checks [24][25][26].

In the early 1980’s, Bertrand Mayer introduced the concept of Design by Contract as an integrated part of the Eiffel programming language [27]. At its core, Design by Contract in Eiffel defines a software contract that specifies the actual behaviour of a software artifact. From this perspective, the purpose of a contract is to ensure that a software module is working according to its contracted behaviour. Bertrand Mayer defines a software contract as the answer to three questions [27]:

1. What does a contract expect?
2. What does a contract guarantee?
3. What does a contract maintain?

Answering each of these questions is done by three corresponding keywords commonly used to specify software:

1. \textit{Requires}: defines what a method is expecting as input.

2. \textit{Ensures}: defines what a method guarantees to return.

3. \textit{Invariance}: asserts unchanging conditions.

Contracts can be placed in interfaces, ensuring the correctness of an implementation. Alternatively, contracts can be placed in methods or in classes. For instance, class invariance has its application to ensure non-changing conditions across all methods and instances of the defining class. However, where to place a contract depends to a certain degree on the detail specified in a software artifact and on how projects are organized. For instance, in distributed software projects, contracting interfaces is relevant to ensure the integration of independently developed sub-systems.
3.1.1 Precondition

A precondition defines precisely what a method is expecting with regard to input, as well as the requirements that must be satisfied before method execution. For instance, a division operator may define the precondition that a divisor must be positive in order to avoid division by zero errors. If any argument does not meet this precondition, an error is thrown and must be handled.

Evidence from combinatorial testing reports that a large proportion of software errors in software projects are triggered by the interaction between one to six parameters. For example, if the first parameter has a specific value, a second parameter must be within a certain range in order to prevent errors otherwise triggered by the control flow of the method. This is a common source of errors if unchecked parameters are passed into the method body [28]. Using preconditions helps address the parameter issue by defining dependencies between parameters as checked preconditions. If an invalid combination of values is passed to the contracted method, the failed precondition actually helps to identify the problem which, in the worst case, would have passed unnoticed.

Even though, input validation and parameter checks are commonly used application for preconditions, further requirements can be checked as well. For instance, a method requires certain resources which means a precondition may perform a resources check prior to method execution. Likewise, if fields are assumed to contain a value, using a precondition allows to explicitly perform a test if a class or field has been initialized correctly. Using preconditions supports precise debugging as any contract violation points directly at the failed precondition and thus the source of the error is known instantly.

In summary, the general concept of preconditions is that any assumptions about conditions required will be stated explicitly at the outset for executing a method. Preconditions can also be verified during compilation as well as during run-time to enforce the contracted preconditions while supporting debugging.
3.1.2 Invariance

An invariant condition defines any state that does not change after initialization and thus is in-variant. In programming, this can be a shared field, for instance containing a network address frequently used amongst several methods or an external defined value required for a calculation. For fields that never change over the entire life time of a class instance, constants are the preferred way of defining these values. However, in some cases, it is required that a certain value may change but must remain unchanged. An invariant check actually performs an equality check if the supposed invariant field still equals its original value at a different time. Thus, invariance checks fail if a supposed invariant field is 1) mutable and 2) its previous value has been unexpectedly modified later. Conversely, immutable fields are invariant by default and do not require additional checks. The importance of invariance checks is significantly higher in concurrency programming in order to prevent oblivious bugs triggered by several threads modifying a shared field simultaneously. For instance, a commonly used invariant check is if a field after method execution still holds the same value as before method execution. This check detects unwanted modifications performed by other threads which usually happens if field access is not strictly ordered amongst threads [27][29].

3.1.3 Postcondition

A postcondition defines precisely what a method is expected to return. A postcondition can only be valid under the assumption of valid input according to its precondition. A postcondition only specifies what to return but not how to reach that goal. For instance, a method calculating a particular classification is expected to return one class for a particular range of the given parameter and another class for the remaining values. A precondition defines the total range of all valid input, for instance a numeric range from one to twelve. The postcondition on the other hand, defines what class must be assigned for certain values of the range. For instance, class A must be assigned to any value smaller or equal to six and class B to all others. The method body defines the actual calculation for the classification. If the actual calculation depends on an external value that may or may not change, an invariant check is applicable to ensure the correctness of the result.
In case of incorrect input, an error of the precondition is thrown whereas an incorrect implementation (that does wrong classification) throws an error triggered by the postcondition. Furthermore, if the assumed external field has changed during the computation, an invariant violation error is thrown. This detailed distinction separates different error sources which is a welcome help during debugging [27][29].

### 3.1.4 Liskov Substitution Principle

The Liskov substitution principle (LSP) is a definition of a sub-typing relation, called (strong) behavioural sub-typing, which is commonly applied in object-oriented programming languages. The principle was first introduced by Barbara Liskov in a 1987 conference keynote considering hierarchies in data abstraction [30]. In 1994, Liskov and Jeannette Wing [31] formulated the principle formally as:

**Definition 3.1.1 (Liskov substitution principle).** Let \( q(x) \) be a property provable about objects \( x \) of type \( T \). Then \( q(y) \) should be provable for objects \( y \) of type \( S \) where \( S \) is a subtype of \( T \) (p. 2 in [31]).

In addition to type hierarchy, Liskov specifies three rules for inheritance of contracts. The first rule says that preconditions can only be weakened in subclasses. The second rule states that postconditions are only allowed to be strengthened in sub-classing. Third, invariants of the super type must be preserved in a subtype. Furthermore, a relevant concrete operation can be substituted for the corresponding operations of all its relevant ancestors (4.2 in [32]).

Consequently, if a programmer violates the rule of only writing weakening inheritance hierarchies for precondition, the LSP causes execution failure of the contracted methods because precondition operators cannot be substituted correctly. The exact failure details depend on how method dispatching is implemented in a programming language.

Liskov’s principle also defines some standard requirements with regards to method signatures that have been adopted in newer object-oriented programming languages such as Scala. In order to ensure the correctness of constraints under inheritance, the LSP requires contravariance of method arguments in the subtype but covariance of return types in the subtype [30][31].
3.2 DBC in Eiffel

Bertrand Meyer introduced Eiffel in 1989 for the construction of reliable software for industry usage. Eiffel supports pre and, postcondition, as well as checks for class invariance. Multiple inheritance in Eiffel follows the Liskov Substitution Principle. The example\(^1\) in listing 3.1 shows an interface defining a contract for an account class. Preconditions are expressed by using the \textit{require} keyword whereas postconditions are expressed by the \textit{ensure} keyword. Even though the example does not contain invariance, there is an invariant keyword available in Eiffel. An Eiffel specific feature is access to the state of a field before a method was applied by using the \textit{old} keyword. The old keyword simplifies comparisons of state changes as shown in this example \cite{invitation-07}\cite{invitation-07}\cite{invitation-07}.

```eiffel
class interface ACCOUNT create
  make
  feature
    balance: INTEGER
    deposit (sum: INTEGER) is
      require
        sum >= 0
      ensure
        balance = old balance + sum
    withdraw (sum: INTEGER) is
      require
        sum >= 0
        sum <= balance - minimum_balance
      ensure
        balance = old balance - sum
    ...
  end
```

\textbf{Listing 3.1: DBC in Eiffel}

Design by Contract in Eiffel is also the foundation of active research by the Chair of Software Engineering at ETH Zurich\(^2\). Three major areas are under active development: contract based automated testing, inferring better contracts and automated fixing of [Eiffel] programs.

\(^1\)http://archive.eiffel.com/doc/online/eiffel50/intro/language/invitation-07.html
\(^2\)http://se.inf.ethz.ch/
Built on Eiffel’s DBC, contracts are used to automate testing of Eiffel software. There are two major challenges involved in complete test automation. The first one is the generation of appropriate input values. The exact problem is to determine the exact size of testing input. If the testing input is too large, the process becomes inefficient, takes too much time and generates unnecessary test-cases. If the testing input is too small, the result is incomplete test coverage and thus potential hidden errors. The second challenge is to decide automatically whether a test-case has passed or failed; this is also known as the oracle problem.

### 3.2.1 Contract inference

Using contracts for testing allows the testing input to be based on the precondition using the postcondition and invariant notion as test-oracles. However, using contracts as oracle raises some problems such as incomplete or even incorrectly written contracts, which may lead to misleading test results. Incomplete contracts lead to insufficient test-case generation, and thus to incomplete test coverage. Incorrectly written contracts, for instance those containing assertions that do not reflect the actual intention of the software, may lead to incorrectly reported bugs of a correct software. Finally, precise fault isolation is challenging as one and the same software error may trigger countless generated test cases to fail.

Addressing the issue of incomplete or incorrect contracts, active research is looking for ways to infer better and sometimes even complete contracts. Ongoing work from Wei et al. provides a tool called AutoInfer that is based on dynamic contract inference [34].

The concept of dynamic contract inference is based on the techniques of inferring specifications by generalizing information collected through actual program execution. The requirement for applying dynamic contract inference is a set of configurations under which the program can be executed correctly. In order to automate the whole process, automated testing gets applied first; next, all all passing test cases are collected and finally, those test cases are used as input for contract inference. AutoInfer applied to basic Eiffel data structure classes such as array, list, queue, stack, set and hash table shows that 75% of the complete postconditions of commands can be inferred fully automatically [34].
3.2.2 DBC and automated testing

Many static and dynamic software analysis tools have been developed in order to reduce time and effort in software testing. Static analysis aims to find software faults by examining the source-code without executing the program [35]. Rutar et al. compared the results of six static source-code analyser tools and their experimental results indicated that none of the tools strictly subsumes another; instead, the tools often find non-overlapping bugs. More significantly, Rutar et al. concludes: "The main difficulty in using the tools is simply the quantity of output" [36].

Dynamic analysis, on the other hand, analyses behaviour of the application during execution time. However, in practice dynamic and static analysis is often combined in order to improve the quality of testing.

Several combinations of static and dynamic analysis exist; for instance JCrasher [37], Check n’ Crash [38] and DSD-Crasher [39] combine both static and dynamic analysis methods in order to detect bugs by forcing an application to crash using random parameters. However, in a comparison of five of such tools, Wang et al. conclude that, "these tools generate tests that are very poor at detecting faults" [40].

More sophisticated testing approaches can be divided into several categories [41] such as mutation testing, metamorphic testing, structural testing or combinatorial interaction testing. Mutation testing involves modifying the source-code in a small way in order to identify weaknesses or blind spots in the test suite [42]. Metamorphic testing [43] can solve the oracle problem, which is defined as "it is difficult to decide whether the result of the program under test agrees with expected result" [44]. Structural testing aims to use a reduced set of tests required to find faults related to the internal structure of the tested software [45][46].

Combinatorial Interaction Testing [3] (CIT) is a black box sampling technique derived from the statistical field of experiment design. It has been used extensively to sample testing inputs for software. The general concept of CIT is defined as a process of five steps [41]. The first step is model identification, followed by test generation and prioritisation. The third step is test execution followed by the fourth step of failure diagnosis. The last step is evaluation and metric calculation based on the test results.
CIT is of particular interest, because the approach has been under active research for over twenty years and has a solid theoretical foundation\textsuperscript{3}. CIT is commonly applied in the aviation industry as well as in government, military and financial industries.\textsuperscript{4} Heller [47] uses an example to show that testing all combinations of parameter values is not feasible in practice and concludes that there is a need to identify a subset of combinations of manageable size. Dalal and Mallows [48] provide a summary of existing combination strategies and their limitations. They also introduce a new approach to software faults in which faults are classified according to how many parameters (factors) need distinct values to cause a fault to result in a failure.

In their discussion on effectiveness, Dalal and Mallows ask the question how many factor values are sufficient in practice? Following this question, Kuhn [28] investigated 365 error reports from two large real life projects and discovered that pair-wise coverage was nearly as effective in finding faults as testing all combinations. This finding was confirmed in a follow up study by Wallace using pair-wise coverage on seven real software projects [49]. These findings suggest that nearly all software errors of the observed software projects were triggered by the interaction of one to six parameters [28][50] as visualised in figure 3.1.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure3_1.png}
\caption{Ratio software faults to nr. of parameters [3].}
\end{figure}

\textsuperscript{3}There are over 50 publications dedicated to test suite generation, a NP-hard problem [41]

\textsuperscript{4}http://csrc.nist.gov/groups/SNS/acts/acts_users.html
In "Combination testing strategies: a survey" (2005) [51] Grindal et al. present the results of investigating 16 different combination strategies from over 40 publications related to Combinatorial Testing. In their conclusion, the authors identified three issues (p. 29 in [51]) that remain largely unexplored:

1. The identification of suitable parameters and parameter values

2. Constraints among the parameter values which are not adequately investigated

3. How to select an appropriate combination strategy for a particular test problem

Applying design by contract to automated testing is a practical way to address exactly these issues. More precisely, a precondition already defines suitable parameters and parameter values as well as constraints between parameters. Furthermore, a postcondition can serve as a test oracle in order to decide if a test has passed or failed. Automated testing in Eiffel uses exactly this approach in order to overcome the issues previously mentioned.

Contract Driven Development (CDD) and AutoTest are two tools that test Eiffel software by automatically using contracts. CDD has been developed by Leitner et al. [52] with the goal of using contracts to generate test cases, and executing these generate tests automatically. CDD is able to replicate failed tests in order to verify if a patch has really resolved the underlying issue. However, the major drawback of CDD is that a program needs to be executed several times by the developer in order to generate a sufficient number of test cases [52]. CDD has since been replaced by AutoTest.

AutoTest is a contract-based random testing tool that fully automates the testing work-flow. AutoTest is fully integrated into EiffelStudio, the main IDE for Eiffel development. Addressing the issue of input generation, AutoTest uses adaptive random testing which means it tries to generate new test inputs that are far away from previous inputs. The main strategy for testing is called stateful testing: this generates new test cases that improve over an existing test suite by purposely violating the dynamically inferred contracts. Stateful test cases are likely to detect new errors, and to improve the accuracy of inferred contracts by discovering those that are unsound [53][54][55].
AutoTest demonstrates its usefulness by experimental evidence. First of all, in a long running random test experiment, over 50% of faults detected by AutoTest were found after the measured branch coverage had not increased any more. The authors of the study concluded that branch coverage is not a good stopping criterion for automated testing [56]. Second, when applied to the EiffelBase 5.6 library, AutoTest was able to automatically detect 98 new bugs\textsuperscript{5} despite the fact that the Base library had already been in industry usage for many years [53]. Third, experiments on 13 data structure classes with over 28,000 lines of code, demonstrate the effectiveness of stateful testing. Stateful testing found 70% of new errors and improved the accuracy of automatically inferred contracts to over 99%, with just a 6.5% additional time overhead [34].

\textsuperscript{5}A bug is defined as preventing the tested classes from working correctly according to its contract
3.2.3 DBC and automated program fixing

Based on the experience with AutoInfer and AutoTest, Yi Wei and Yu Pei are working on automated program fixing in Eiffel. They have already developed two tools that work on Eiffel source code. AutoFix-E, based on behavioural models of a class, and AutoFix-E2, introducing evidence-based techniques. The key insight behind both tools is to rely on contracts present in the software to ensure that the proposed fixes are semantically sound. AutoFix-E requires several steps, visualized in illustration\(^6\) 3.2, in order to deduce a fix automatically. In the first step, AutoTest generates and executes test cases based on contracts. The output contains all passed as well as failed test cases. The next step analyses all failing tests which results in a fault profile.

![Automated fix process in AutoFix-E.](http://se.inf.ethz.ch/research/autofix/)

The fault profile defines the necessary failing condition showing why a fault is happening. Next, a finite-state abstraction is created by analysing only the passing tests and a profile describes how different routines affect different queries. The finite state abstraction contains all required information to determine how to change the system state to avoid the necessary failing condition. AutoFix uses the fault profile as well as the finite-state abstraction in order to deduce potential fixes. Next, it reruns the tests to validate those candidates, which means a fix is only reported if passing all tests [33][57]. In an experiment conducted in February 2010, AutoFix-E was applied to 42 faults found in production Eiffel software, and it was able to fix 16 (38\%) of those faults [57].

---

\(^6\)Src: [http://se.inf.ethz.ch/research/autofix/](http://se.inf.ethz.ch/research/autofix/)
AutoFix-E2 (AF2) uses an evidence-based approach. AF2 uses a scoring and ranking based work-flow, as shown in figure 3.3, to determine a fix. In the first step, AF2 uses AutoTest as well to generate passing and failing test cases for each class.

![Figure 3.3: Evidence based fix work-flow in AutoFix-E2](image)

Next, all expressions from the class are extracted. From there, AF2 computes the expression dependence score edep, the control dependence score cdep and the dynamic score dyn. The edep score measures the syntactic similarity between expressions. The control dependence score cdep measures how close the expressions are on the control-flow graph. The dynamic score dyn between expressions measures how often an expression is mentioned in failing rather than passing test cases. All the scores are combined together into the ”fixme” score which determines a global ranking of all expressions. In the next step, possible fixing actions are enumerated for the expressions with the highest fixme score. Potential fixes are generated by by injecting the fix actions into the faulty routine. Only fixes that pass all the regression test suites are considered valid [4].

Experiments with AutoFix-E2 include two data sets. In the first one, AutoFix-E2 was applied to 15 faults from EiffelBase. AutoFix-E2 produced correct fixes for 12 of these faults, some of which are beyond the capabilities of AutoFix-E. The second data set includes 5 faults from a library manipulating text documents developed as a student project. AutoFix-E2 fixed 3 of these faults [4][57].

In summary, Design by Contract in Eiffel is the foundation for a fairly sophisticated tool ecosystem enabling a certain level of testing, contract inference and program fixing automation for software written in Eiffel.
3.3 DBC in Java

3.3.1 Overview

From Java 1.4 onwards, Java provides an `assert` keyword (JSR 411) supporting rudimentary assertion checking. The major limitation of Java’s `assert` keyword is its run-time behaviour of only throwing a generic assertion violation exception in case of a failed assertion check. Furthermore, assertions checks in Java are disabled by default during runtime and can only be activated by using JVM flags. Even Oracle’s official ”Programming With Assertions” guide\(^7\) discourages the use the of assert keyword in preconditions. Instead, explicit checks that throw specified exceptions are encouraged. Furthermore, the guide illustrates some basic ways on how to write pre and postconditions using Java’s built in facilities.

Because of Java’s limited support for Design by Contract, several projects have emerged over time in order to supply better contracts for Java. I have identified and reviewed fifteen DBC projects for Java. The following section categorizes these projects into three groups, based on how contracts are written.

The first group uses the Java Modeling Language (JML) to write contracts. JML was originally used to write specifications for Java projects. Section 3.3.2 summarizes those (few) tools requiring contracts to be written in JML. The second group (section 3.3.3) uses plain Java and dependency injection (DI) to connect contracts and classes together. Using dependency injection allows inheritance for contracts and implementation as well as a clear separation of concerns. The third and largest group of tools uses Java’s annotation to define contracts, as summarized in section 3.3.4. Additionally, a small group of non-active DBC tools have been identified and briefly summarized in section 3.3.5.

\(^7\)http://docs.oracle.com/javase/1.4.2/docs/guide/lang/assert.html#usage-control
3.3.2 JML based contract tools

Name: JaSS: Java with assertions
First Release: 1999
Last Update: 2007/Aug
Total Downloads: N/A
Project: http://csd.informatik.uni-oldenburg.de/~relatedjass/
Publication: Java with assertions.

Jass is based on oVal, the Object Validation framework and was developed by the working group of correct system design at the Department of Computer Science at Oldenburg University, Germany. Until version 2, Jass used its own specification language but beginning with the development of version 3, Jass started to use JML as a replacement. Despite being developed for JDK 1.4, Jass offers a comprehensive set of functionality such as pre and postconditions and invariant checks for methods, as well as rescue and retry statements. A newer version, using Java 5 annotations, is ModernJass which is summarized in section 3.3.4.
3.3.3 Dependency injection based contract tools

**Name:** C4J-Team Contracts Framework for Java  
**First Release:** 2012/March  
**Last Update:** 2007/Aug  
**Total Downloads:** 68  
**Project:** [https://github.com/C4J-Team/C4J#readme](https://github.com/C4J-Team/C4J#readme)

The C4J-Team contracts framework is built on the idea of using inheritance to declare contracts and dependency injection to inject a contract into an implementation. A contract is defined in a Contract Class which may extend the target super class for convenience and better refactoring support. A contract must be satisfied not only by the type for which it is declared, but also by all classes that are extending or implementing this type. Illustration 3.4 shows the inheritance hierarchy. For better development support, an eclipse plugin is under development\(^8\).  

![C4J contract inheritance](image_url)

**Figure 3.4:** C4J contract inheritance

Using the simple cash terminal class in listing 3.2 as an example, a contract class needs to be annotated to the implementation using the @ContractReference annotation. The contract in listing 3.3 overrides the methods and adds pre and postconditions that will be injected into the account class.

\(^8\)https://github.com/C4J-Team/C4J-Eclipse-Plugin
Chapter 3. Design by Contract

1 @ContractReference(CashTerminalContract.class)
2 public class CashTerminal {
3   public void withdraw(int amount) {
4     // impl
5   }
6   public int getBalance() {
7     // impl
8   }
9 }

Listing 3.2: C4J-Team usage example

1 public class CashTerminalContract extends CashTerminal {
2   @Target
3   private CashTerminal target;
4
5   @Override
6   public void withdraw(int amount) {
7     if (preCondition()) {
8       assert amount > 0;
9     }
10    if (postCondition()) {
11       assert target.getBalance() == old(target.getBalance()) - amount;
12    }
13  }
14 }

Listing 3.3: C4J-Team contract
Chapter 3. Design by Contract

Name: C4J: Design By Contract for Java  
First Release: 2006/March  
Last Update: 2011/April  
Total Downloads: 1.921  
Project: http://c4j.sourceforge.net

C4J claims to be a non-intrusive DBC tool for Java that makes it easy to add contract verification of instances and methods of any Java class or interface. C4J provides a comprehensive set of advanced functionality such as concurrency support and contract refinement according to the Liskov substitution principle (see 3.1.4).

Also, it is possible to register loggers which are notified when contracts are verified, e.g. to support statistics gathering. Contracts are defined in separate Java classes injected into the actual implementation. Programming conventions are very similar, as the example in listing 3.4 below illustrates.

```
1 public class DummyContract
2  extends ContractBase<Dummy>{
3     public DummyContract(Dummy target){
4         super(target);
5     }
6     public void classInvariant() {
7         assert m_target.m_stuff != null;
8     }
9     public void pre_addItem(Object item){
10        super.setPreconditionValue("list−size",
11           m_target.m_stuff.size());
12     }
13     public void post_addItem(Object item){
14        assert m_target.m_stuff.contains(item);
15        int preSize = super.getPreconditionValue("list−size");
16        assert preSize == m_target.m_stuff.size() − 1;
17        assert m_target.m_stuff.size()
18           == super.getReturnValue();
19     }
20 }
```

Listing 3.4: C4J contract
Chapter 3. Design by Contract

Name: C4J: SpringContracts
First Release: 2006/Nov
Last Update: 2007/Jan
Total Downloads: 1.152
Website: http://sourceforge.net/projects/springcontracts
Project: http://springcontracts.sourceforge.net/home.html

SpringContracts has been developed by Mario Gleichmann and incorporates DBC into the Spring framework. Using Spring allows a flexible configuration of how to react to contract violations. For further customization, a custom contractViolationHandler can be written and injected into the application context using Spring. Writing contracts in SpringContracts first requires the configuration of the application context by using Spring’s xml configuration style. Using the default settings run the framework using annotations as specification language and throw ContractViolationExceptions if a violation of a contract element is happening. Alternatively, either BeanShell or Java can be used as a specification language for contracts. Once the context is defined, any class using contracts needs to be registered as well, so that spring can advice the annotated aspects. Contracts are defined in interfaces to keep implementations free of specification details. Listing 3.5 shows an example Spring contract.

```java
@Invariant(condition="this.capacity > 0 and this.size <= this.capacity")
public interface Stack {
    @Postcondition(condition="return >= 0")
    public interface Stack {
        @Precondition(bindArgs="arg1=element", condition="!empty element")
        @Postcondition(bindArgs="arg1=element",
            condition="this.size == old: this.size + 1 and this.top == element")
        public void push(Object elem);
        public Object getTop();
    }
    @Postcondition(condition="(this.size > 0) ==> (!empty return)"
    public Object getTop();
}
```

Listing 3.5: SpringContracts example

Dependency Injection is used so that a contract violations handler gets injected into the application context by using Spring’s DI facilities. However, as the last release dates back to 2007, it’s not clear to me if the SpringContracts project is still active.
3.3.4 Annotation based contract tools

Name: C4J: chex4j
First Release: 2010/Aug
Last Update: 2011/Apr
Total Downloads: 166

Chex4J is a DBC framework inspired by contract4j that uses annotations to express pre and post conditions on method calls. A sample contract is shown in listing 3.6. Chex4J uses Javassist to redefine the annotated classes by translating the annotations into bytecode and injection the bytecode into the corresponding class prior to class loading.

Chex4j supports an online mode as well as an offline mode. The online mode is implemented by using a Javaagent which instruments classes at load time to provide assertion checking either for running an application or to enable IDE’s for development. The offline mode, on the other hand, is implemented as a main method that can be used by common Java build tools such as Ant or Maven.

```java
@Contract
public class SimplePublicBankAccount {

    @Post("amount.doubleValue() >= 0.0d")
    public SimplePublicBankAccount(BigDecimal amount) {
        this.balance = amount;
    }

    @Post("$_.doubleValue() >= 0.0d")
    public BigDecimal getBalance() {
        return this.balance;
    }
}
```

Listing 3.6: Chex4j example

However, despite its well though concepts, Chex4J clearly misses documentation; more advanced functionality such as invariance checks; and most importantly, a non-intrusive setup, as deploying java-agents in production environments is not always a desired option. The absence of any information considering inheritance and the handling of contract violation makes the adoption of chex4j especially impractical.
Contract4J5 is a project of Aspect Research Associates\textsuperscript{9} founded by Dean Wampler and aims to provide DBC functionality. Contracts are written in annotations and it uses ”prebuilt” aspects for contract evaluation during run-time. Listing 3.7 shows a contract example using pre and postcondition.

```
@Contract
public class SearchEngine {
    @Pre
    @Post("$return != null && $return.isValid()")
    public PhoneNumber search(String first, String last, Address stAddress) {
        PhoneNumber result = doSearch(first, last, stAddress);
        return result;
    }
}
```

**Listing 3.7:** Contract4J5 example

The level of documentation is good, and a detailed description for each annotation as well as inheritance exists. As the only DBC project so far, Contract4J5 provides a clear statement regarding the performance impact of using it:

"Unfortunately, the current runtime overhead of C4J is prohibitive, making it difficult to use continuously on large projects"\textsuperscript{10} Some performance measurements exist\textsuperscript{11} but these are of limited value as they compare different techniques of aspect weaving available for C4J but lack a ”base” performance of the application run-time without C4J applied. Therefore the general performance overhead of C4J is unclear. However, the ”prohibitive” performance penalty refers to the usage of load-time weaving which is known for its poor performance.

\textsuperscript{9}http://aspectresearchassociates.com/
\textsuperscript{10}http://polyglotprogramming.com/contract4j/c4j5# todo
\textsuperscript{11}http://polyglotprogramming.com/contract4j/c4j5# how
Modern Jass started as the master thesis of Johannes Rieken and is a modern version of the Jass project using annotations. More precisely, Modern Jass uses Java 5 annotation to specify contracts; a pluggable annotation Processor to validate contracts, and bytecode instrumentation to enforce contracts at run-time. Listing 3.8 shows a class defining a precondition on a main method.

```java
public class Bar {
    @Pre("args.length % 2 == 0")
    public static void main(String[] args) {
        System.out.println("Hello, " + args.length);
    }
}
```

Listing 3.8: Modern Jass example

Modern Jass ships with a large set of predefined annotations to define constraints such as length, min, max or range of a value. Furthermore, with regard to method level precondition, postcondition, invariance and "method purity", absence of side-effects, can be annotated. The modern Jass project has been suspended and has finally led to the Contracts for Java project.
Cofoja is a rewrite of ModernJass by a team of Google developers together with Johannes Rieken, the original developer ModernJass. Cofoja allows annotation of pre, postcondition and invariant. Similar to Chec4j, Cofoja requires a Javaagent to work correctly. Contracts are written directly into class or interfaces. An example is provided in listing 3.9. Furthermore, contracts can be written in two different forms, multi-clause and simple form. A multi-clause allows writing several expressions per annotation (@invariant in the example) whereas the simple form only allows writing of only one expression per annotation. (@Ensures in the example)

```java
@Invariant({
    "name != null",
    "!name.isEmpty()"
})
class Person {
    private String name;
    @Ensures("result == name")
    public String getName() {
        ...
    }
    @Requires({
        "newName != null",
        "!newName.isEmpty()"
    })
    @Ensures("getName() == newName")
    public void setName(String newName) {
        ...
    }
}
```

**Listing 3.9:** Cofoja example
Cofoja does support inheritance for both, classes and interfaces, and follows the Liskov substitution principle (see 3.1.4).

Cofoja has three important differences compared to other DBC frameworks. The first one is that it is disabled by default for performance reasons. The developers admit a certain loss in performance\textsuperscript{12} but do not quantify this further. Addressing the performance issue, Cofoja can be globally enabled, or selectively, for instance, just for a particular library.

The second difference is the @ThrowEnsures annotation which defines a condition when an exception is thrown. The third major difference is the addition of contract annotations to JavaDoc, which is a convenient way for improving documentation.

\textsuperscript{12}http://code.google.com/p/cofoja/wiki/FAQ
The oVal, object validation, framework offers not only contract functionality but also in-depth object inspection for any kind of Java objects. Custom constraints can be expressed as annotations, as plain Java classes, as XML or by using scripting languages such as JavaScript, Groovy, BeanShell, OGNL or MVEL. Furthermore EJB3 JPA annotations as well as Bean Validation (JSR303) annotations can be translated into equivalent oVal constraints; this is convenient as object validation can be improved by changing the validation framework without modifying already existing constraints or contracts.

Design by Contract is supported by precondition, postcondition and invariance checks. Furthermore, the exact checking behaviour can be configured. For instance, object validation can be enforced before method execution which is of particular value as no roll-back operations are required to undo partial modifications of a failed method call caused by a constraint violation during execution.

Conversely, object validation can also be configured after method execution which is valuable when lazy initialized values are used. However, if a constraint violation happens, the method will have already been executed, and thus any changes will need to be rolled back manually. This is particularly harmful in a multi-threading application and requires careful lock handling to avoid unwanted side effects.

Writing contracts in oVal is efficient and also very powerful as many useful predefined constraints are provided. If the "applyFieldConstraintsToSetters" parameter is used, as shown in listing 3.10, this will be the only fields that need to be annotated and the validation will then be enforced for all setters of the corresponding fields.

Compared to all other DBC Tools, oVal appears to be the most complete and mature project available to date.
Chapter 3. Design by Contract

```java
@Guarded(applyFieldConstraintsToSetters=true)
public class Person {

  @NotNegative
  private int age;

  @Min(5)
  private String name = "";

  @Length(min=5, max=5)
  private String zipCode = "";

  public void setAge(int age) {
    this.age = age;
  }

  public void setName(String name) {
    this.name = name;
  }

  public void setZipCode(String zipCode) {
    this.zipCode = zipCode;
  }

  ...
}

Listing 3.10: oVal example
3.3.5 Inactive projects

Name: Barter - beyond Design by Contract
Total Downloads: 822
Barter allows contracts to be defined as aspects in JavaDoc comments. Essentially, Barter is a code generator for AspectJ, implemented as an xDoclet task. The development stagnated shortly after the first release, and even the project page officially states that the project is obsolete and not maintained any more.

Name: JoC: Java on Contracts
Total Downloads: 33
Project: http://code.google.com/p/java-on-contracts
This project has been suspended in favour of developing the C4J-Team project as successor

Name: JavaDBC for Java
Total Downloads: N/A
Project: http://code.google.com/p/javadbc
It is not clear if this project has ever been used as there is no release available for download.

Name: JContractor
Total Downloads: 4.354
Publication: "jContractor: Bytecode Instrumentation Techniques for Implementing Design by Contract in Java." DOI:10.1016/S1571-0661(04)80577-2 [60]
JContractor follows the approach of using name conventions to add contracts; either within a class, or as a separate class file that extends the implementation class. All contracts are written in plain Java code and thus follow Java’s convention for inheritance and polymorphism. However, after the last release back in early 2003, no further development activities have been recorded.
Chapter 3. Design by Contract

**Name:** JcontractS: Java Contract Suite
**Total Downloads:** 1.796
**Project:** http://jcontracts.sourceforge.net/index.html
JcontractS is a resurrection of the iContract tool originally developed for JDK 1.2 by Reto Kramer of Reliable Systems. John Swapceinski started a newer version by creating iContract2 by decompiling the original iContract class files. Johan Stuyts finally created the Java Contract Suite based on iContract2. Since the second release in 2008, no further development effort is observable.

**Name:** iContract Plus
**Total Downloads:** 294
**Project:** http://sourceforge.net/projects/icplus/
The aim of this project was to "increase the usability of the iContract tool"\(^{13}\)

**Name:** JMSAssert
**Total Downloads:** N/A
**Project:** http://www.mmsindia.com/DBCForJava.html
JMSAssert is a DBC tool developed by a company located in Thiruvanmiyur, the southernmost part of the Indian Peninsula. Neither a download nor any release information are available. However, as the webpage was last updated in 2007 and still refers to iContract which has not existed for years, it is most likely that JMSAssist is not used at all.

\(^{13}\)Source: http://sourceforge.net/projects/icplus/
3.4 DBC in Scala

In Scala pre and postcondition checks are integrated into the language itself. Contract validation is done by the compiler but a developer can add specific error messages to a pre or postcondition to simplify debugging. Furthermore, the Scala compiler shows the specified error message first followed by the stack-trace; in most cases, a developer can fix the contract violation very quickly. Writing a contract in Scala requires the following programming steps [61]:

1. A trait that is used as an interface
2. A trait that defines the specification by extending the interface
3. Another trait that implements the specification and extends the interface
4. A class, that mixes the implementation and the specification together
5. A companion Object to provide static access to the contracted class.

According to M. Odersky [61], the added overhead is motivated by gaining noticeable advantages such as contract inheritance, modular combination of contracts and implementation. For small use cases, all traits and classes can be written in just one file.

However, while writing a set of small example contracts in Scala, certain shortcomings in Scala’s current DBC implementation have been identified. First of all, pre as well as postcondition can only handle one Boolean expression as parameter. For example “requires(x > 10)” is valid, whereas “ensuring(ret == sum/20*3)” is not possible since the latter contains more than one expression. This particular issue is a strong limitation in terms of using Scala’s DBC in practice.

Second, in-depth Object inspection is not feasible since it is limited to Boolean expressions. For instance, it is possible to check whether a parameter is null, or is within a particular range but inspecting whether a string contains one or more particular characters is not possible. Case classes, that are Scala’s equivalent to Java Beans, can be inspected since all properties are publicly accessible. On the other hand, pattern matching on object properties is not possible. Finally, the last limitation is the lack of invariant checks in Scala.
Chapter 3. Design by Contract

The Scala Style Guide discourages the usage of mutable fields of any kind. Instead, in Scala the preferred way is to modify only parameters and use only immutable fields, which follows best practice in functional programming. Consequently, invariant checks are, in theory, less of an issue but in practice they are still required, mainly because of poor programming.

Example DBC in Scala

Using the introduced contracting approach, I have written an interface, a specification and an implementation to illustrate DBC in Scala. Figure 3.5 gives an overview of the example and a complete listing of the source code is located in Appendix A. Figure 3.5 shows that implementation and specification are traits (orange) while get are mixed together in a class (red) called Account. The account class is also represented through a companion object (green) in order to allow static access to the class.

![Diagram](image)

**Figure 3.5:** Scala DBC example

The interface, shown in listing 3.11 defines four methods, openAccount, credit, withdraw and printBalance. Both, implementation and specification traits inherit from the interface.
Listing 3.11: Interface for Scala DBC example

Listing 3.12 shows an excerpt of the trait defining the specification for the account example. The specification defines preliminary empty methods by performing a "super" call to the signature of the interface. This is valid because the actual composition needs to supply an implementation of the interface in order to dispatch the super call of the specification correctly. Because of that, the specification is free of implementation details and only defines contracts for each method by adding pre and postconditions.

Listing 3.12: Contract for Scala DBC example

The implementation on the other hand only overrides the signatures of the interface by adding the actual implementation, as shown in listing 3.13. The compactness of the implementation is a direct result of having moved all domain specific constraints into the corresponding specification class.
Chapter 3. Design by Contract

Listing 3.13: Implementation for Scala DBC example

```scala
/** implementation */
protected trait Account_C4S_Impl extends AccountI {
  var accBalance = 0
  override def openAccount(b: Int) { accBalance = b }
  override def credit(amount: Int) { accBalance += amount }
}
```

Next, implementation and specification traits are mixed together in a class called Account. The account class contains, as shown in listing 3.14, all methods of the interface but if a method gets called, the call first gets checked by the corresponding specification and is then dispatched to the actual implementation. For instance, opening an account with a negative initial balance throws the corresponding contract violation error showing the custom error message defined in the contract. Furthermore, the class account is also represented through a companion object (green) enabling static access to the class. Using mixins to combine contracts and specifications has the benefit of improving cohesion as each trait has just one specialized task and only the combining class contains all required dependencies. Also, either wrong implementation or incorrect specification are simple to correct since only the corresponding trait needs either to be corrected or replaced.

Listing 3.14: Implementation for Scala DBC example

```scala
/** Companion object */
object Account extends Account
/** Mixing class */
class Account extends Account_C4S_Impl with Account_C4S_Spec
```

Extending this modular composition allows the re usage of all three components, the interface, the specification and the actual implementation. For instance, a subclass "Student-account" may extend the general account specification as well as the actual implementation. Since the basic operations such as credit and withdraw are already defined, only additional requirements need to specified and applied to the new student account.
3.5 DBC in ADA

Ada is a structured, statically-typed, imperative, and object-oriented programming language originally designed in the late 1970’s for the U.S Department of Defense (DoD) for the purpose of proving a reliable foundation for military applications. The name Ada is credited to Ada Lovelace\footnote{1815-1852} who is considered to be the first programmer in history\footnote{http://www.guardian.co.uk/technology/2012/dec/10/ada-lovelace-honoured-google-doodle}. The Ada 2012 standard\footnote{ISO/IEC 8652:2012} defines a contract-based programming model fully integrated into the language.

Similar to Eiffel, Ada supports pre and postconditions. Unlike many other languages, Ada also combines type-invariance with subtype predicates that can be applied to type and subtype declarations. Despite the similarity to existing contract implementations, Ada uses a fairly sophisticated implementation that advances the overall usefulness of contracts.

Pre and Postcondition

Preconditions in Ada can be defined on function as well as on class level. Each precondition is defined as an aspect which is linked to the applied item using the \texttt{with} keyword. The stack example in listing 3.15 shows two functions \texttt{push} and \texttt{pop} illustrates how the \texttt{with} keywords is used to link function and contract. Also, using functions (i.e. \texttt{Is\_Empty}) for checking conditions (line 5 - 6 in listing 3.15) makes contracts in Ada very compact yet powerful in expressiveness.

Pre- and postconditions are controlled in Ada by the same mechanism as assertions, which means these can be switched on and off for run-time. Also, Ada distinguishes between method and class pre and postconditions. Class wide postconditions and method pre- and postconditions are checked in the body of the defining class. On the other hand, class wide preconditions are checked in the class where a client is performing the actual call to the contracted class.

This distinction may surprise but the underlying rationale is based on Ada’s dynamic method dispatching which determine the defining class of a method call dynamically during run-time. Listing3.16 gives an example for method dispatching.
Chapter 3. Design by Contract

Listing 3.15: Ada contract example

Assuming that several classes implementing one interface which defines the method "OrderMeal" exist, the example is used to illustrate contracts and dynamic dispatching in Ada. The actual class wide precondition needs to be checked at the point of call to ensure the correct implementation gets checked during method invocation. Listing 3.16 below illustrates this mechanism in Ada.

Listing 3.16: Ada dispatching example

Ada supports contract inheritance by following the Liskov Substitution Principle (see 3.1.4). Consequently, if a programmer is violating the rule of only writing weakening inheritance hierarchies for precondition, method dispatching will fail because of the operation substitution which, if violated, does not guarantee that the triggered precondition implicitly covers the constraints of the actual method call. Therefore, violating the substitution principle will certainly cause the failure of otherwise correct contracts.17

Types and predicates

Unlike most programming languages, Ada does not have pre-defined ranges for data types. Instead only abstract data types such as "Integer" or "Array" exists and a programmer needs to specify custom types by extending an abstract type, defining its range and constraints. Only these custom types can be used within the defined scope. For example, listing 3.17 defines a new custom type Bit Vector with sixteen elements (0 ...15) each of Boolean type. Only this custom type can be used within the defined program scope. If other types are required, they must be declared first.

There are two other facilities concerning types and subtypes in Ada. One is known as type invariants and these describe properties of a type that remain true and can be relied upon. The second is known as subtype predicates which extend the idea of type constraints. An example of an ordinary type definition is shown in listing 3.17:

```ada
1 type Bit Vector is array (0 .. 15) of Boolean
2   with Component_Size => 1;
```

Listing 3.17: Ada type example

Extending the idea of constrains on sub-types further, Ada defines two kinds of predicates, static as well as dynamic ones. As the names already indicate, a static predicate must have an invariant constraint that holds true for all possible values of the specified type. Dynamic predicates on the other hand contain functions that will be dynamically checked. Listing 3.18 gives three examples of sub-type predicates.

The first example illustrates the usage of the modulo function to evaluate if an integer value is even or uneven. The second example (line 5) showcases a static check if an enum value of type month (line 3) contains one of three possible choices.

The third example (line 10) illustrates a dynamic predicate that calls function (line 8) for evaluation which adds additional power of expressiveness to type predicates.
Chapter 3. Design by Contract

1  subtype Even is Integer with Dynamic_Predicate => Even mod 2 = 0;
2  type Month is (Jan, Feb, Mar, Apr, May, ..., Nov, Dec);
3  subtype Winter is Month
4       with Static_Predicate => Winter in Dec | Jan | Feb;
5  subtype SmallPet is Animal
6       with Dynamic_Predicate => Is_SmallPet(SmallPet);
7  function Is_SmallPet(X: Animal) return Boolean;

Listing 3.18: Ada type predicate example

Type invariance

Type invariance only applies to all values that a type defines whereas type constraints and predicates are generally used to specify a valid subset of values. To clarify the difference further, type invariants describe general properties of the type itself whereas type constraints specify the actual values a type provides. Listing 3.19 illustrates the difference by using a stack as an example.

The function ”Is_Unduplicated” in line 6 has to be written to check that all values of the stack are different. The invariant on a private type T is checked when the value can be changed from the outside world of the class. This is the case after default initialization of an object of type T, after a conversion to type T or after a call that returns a result of a type T. The checks also apply to subprograms with parameters or results whose components are of type T. In the case of the stack example, the invariant function call ”Is_Unduplicated” will be checked when a new object of type Stack is declared each time Push and Pop are called.

Both invariance and predicates on sub-types are valuable tools enabling early detection of preventable type errors during compile time checks.
Chapter 3. Design by Contract

Listing 3.19: Ada type invariant example

3.6 Analysis

I evaluated fifteen different design by contract tools for Java by forming eight categories according to the underlying approached of the tool. As required, other programming languages are used as example if no equivalent tool for Java exists. The used categories are:

1. Annotations
2. Specification language
3. Dependency injection
4. Meta language
5. Naming conventions
6. Mixed expressions
7. Language additions
8. Language support

Table 3.6 summarizes the results of the comparison and gives at least one example for each approach. With the exception of language additions, meta languages and language support, all tools mentioned in the table are presented in section 3.3 in more detail.
### Figure 3.6: Comparison of DBC approaches.

<table>
<thead>
<tr>
<th>Approach</th>
<th>Example</th>
<th>Pro's</th>
<th>Con's</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Annotations</td>
<td>Modern Jass</td>
<td>Simple to write</td>
<td>No direct access to fields. Requires runtime instrumentation Limited in expression</td>
<td>Annotations have already proven to be too limited for practical DBC.</td>
</tr>
<tr>
<td>Specification</td>
<td>JML Spec#</td>
<td>Powerful</td>
<td>Complex</td>
<td>Both, JML and Spec# are rarely used outside academic research.</td>
</tr>
<tr>
<td>language</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dependency</td>
<td>Contracts for Java</td>
<td>Simple to integrate and Powerful.</td>
<td>Tool support</td>
<td>Only one project is known to me that actually uses this approach.[2]</td>
</tr>
<tr>
<td>injection</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Naming conventions</td>
<td>jContractor</td>
<td>Compiler support</td>
<td>Error prone in case of smallest name mismatch</td>
<td>It appears that this approach is not actively used any more.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>static checking</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Exception handling</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mixed expressions</td>
<td>Oval supports DBC</td>
<td>Very flexible</td>
<td>Complex tool support and limited expression power due to the different forms of expressing contracts.</td>
<td>Oval appears to be used in practice</td>
</tr>
<tr>
<td></td>
<td>notated in annotation, xml and pojo</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Meta language</td>
<td>PACT</td>
<td>Complex and error prone</td>
<td>No compiler support for static verification,</td>
<td>Same as JML, hardly used anywhere outside (limited) academic research.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>translation into a real programming language</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Language additions</td>
<td>Code Contract (Microsoft)</td>
<td>Static checking Multi-language support</td>
<td>Tool support still in development.</td>
<td>Code contracts is a practical alternative to the rarely used Spec#</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Language support</td>
<td>Eiffel, ADA, D, VDM</td>
<td>Static and dynamic checking Automated verification</td>
<td>Requires higher skills during development.</td>
<td>Eiffel has proven that DBC is working on Industry scale projects.</td>
</tr>
</tbody>
</table>
The results from table 3.6 have brought me to the conclusion that none of the existing tools and approaches is flexible, yet powerful enough to provide a foundation for automated test generation. Namely, the shortcomings are:

1. Contract violation always terminates the application
2. No logging support
3. No configuration support
4. No concurrency support
5. No function evaluation support

All current DBC implementation have the same assertion handling which means, any contract violation throws an exception and terminates the application. During development, this is useful in order to support debugging. However, in automated testing, this is not an option, instead such violations needs to be logged for later analysis. For instance, in production systems, a silent alert combined with detailed logging would be the preferable choice to keep the system running. Currently, there is not one single DBC implementation allowing a flexible configuration of contract enforcement.

Logging support is missing as well which means, if logging is required, it needs to be done separately. This is particularly important as it means, contract checks cannot be logged by default thus makes it more difficult to debug.

Furthermore, there is a total lack for flexible DBC configuration on a per project, per class, per package or even per object base. This is particularly important since software projects are often developed in modules that evolve separately. Having different configurations for different modules implies a tailored support for specific requirements for each module. However, none of the investigated tools provide rudimentary configuration support.
Concurrency support has two important aspects that needs to be considered. One aspect is the concurrent evaluation of contracts which may lead to performance gains. The second aspect is contracts dedicated for ensuring correct concurrent program execution. None of the investigated tools support either way of concurrency.

Finally, functions for complex evaluations are not supported by nearly all DBC tools. The only exception us dynamic predicates in Ada (see 3.5). Instead of passing just a boolean expression to pre or postcondition, using a function as parameter allows a compact yet powerful way to express complex checks that are re-usable.
"If I were to pick a language to use today other than Java, it would be Scala” – James Gosling

4.1 Overview

Scala is a new programming language that unifies functional and object orientated programming in one statically typed language. Conceptually, Scala has a uniform object model so that every value is an object and every operation is a method. In Scala, methods are first-class values and therefore objects. This follows the concept of pure object orientation already known from languages such as Smalltalk. Scala also includes an interactive interpreter, called REPL, which is unusual for a statically typed language as interactive interpreters are more common amongst dynamically typed languages such as Python. Scala’s REPL also allows interactive debugging, which makes it a useful tool for ad-hoc experiments in Scala.

As a multi paradigm language, Scala combines several existing programming concepts with a concise yet powerful syntax. Concepts such as mixin composition are known since the early 1990’s but Scala does it in a different way. For instance, Scala combines traits, abstract type members and explicit self-type annotation for composable mixin compositions without any static reference between components [62].

---

1 Scala’s first release was in June 2004
2 http://www.smalltalk.org/versions/ANSIStandardSmalltalk.html
3 REPL stands for Read-Evaluate-Print Loop
4 http://docs.python.org/2/tutorial/interpreter.html
According to Martin Odersky, who is the inventor of Scala, there are two principal forms of abstraction in programming languages. The first one is parameterized types and the second one is known as abstract members. The principal of parameterized type abstraction is typically used in functional languages, whereas abstract members are commonly used in object orientation [63][64]. Java supports parameterized types for values and abstract members for operations. Scala, on the other hand, supports both styles of abstraction which means types and values can be parameters and members. Types can be passed as arguments (generics) or can be defined as so called abstract type members. The same is true for values which can be parameters as well as abstract members [63].

The next sections provide a comprehensive description of Scala and the involved concepts. As the results in chapter 5 are based on rather advanced functionality of Scala, the following sections are necessarily verbose in an attempt to elaborate the required background for understanding the methodology used. Also, because of the high level of interconnection between all the different concepts, several cross-references are placed as required for understanding. Beginning with a summary of Scala’s syntax in section 4.2, section 4.3 describes Scala’s unified object model. Scala’s type system is illustrated in section 4.4, followed by class linearisation which is summarized in section 4.6. Modular mixin composition and the cake pattern are covered in section 4.7. Details about Scala’s functional programming are covered in section 4.9 which provides insights into anonymous functions and higher ordered functions as well as higher kinded types. Finally, concurrency in Scala, in particular composable futures and actors are covered in section 4.11.
4.2 Scala syntax

Starting with the basics, Scala borrows syntax for some general concepts such as control structures from Java. Despite these basic similarities, several differences between Java and Scala exist. Using the example\(^5\) of a simple for-loop as shown in listing 4.1, the first difference is Scala’s `object` declaration in line 12 which declares a singleton object. Scala has a `class` construct as well which unlike `object` allows the creation of several instances. Java uses static classes for single instances. Scala uses the `id : type` syntax for variable and parameter declarations whereas Java uses the `type : id` syntax. Also, Scala distinguishes immutable values from mutable ones by using the `val id : type` syntax (line 15) to indicate that a field or parameter holds an immutable value. Java uses the `final` keyword for that. Mutable fields are defined as `var id : type` in Scala to indicate that a field holds a variable. Java has no special syntax for this case since all variables are mutable by default in Java unless declared `final`\(^6\)(4.1 in [65]).

```
// Java
class PrintOptions {
    public static void main(String[] args) {
        System.out.println("Options selected: ");
        for (int i = 0; i < args.length; i++)
            if (args[i].startsWith("-"))
                System.out.println(" "+args[i].substring(1));
    }
}

// Scala
object PrintOptions {
    def main(args: Array[String]): Unit = {
        println("Options selected:")
        for (val arg <- args)
            if (arg.startsWith("-"))
                println(" "+arg.substring(1))
    }
}
```

Listing 4.1: Simple for-loop in Java and Scala

\(^5\)(1 in [63])

\(^6\)
Scala requires that all definitions start with the `def` keyword; In the example used `def main` starts a method definition. Semicolons are optional and not required to mark the end of a statement whereas Java enforces the usage of semicolons [63][65]. Following Scala’s uniform object model (next section), an array of Type T is consequently written as `Array[T]` and accessed through methods, for instance `val x = arr(i)`. In this respect, Scala differs noticeably from Java’s special array syntax [63][65].

Return types declaration is optional and is placed after a method’s signature. In the example, main’s return type is `Unit` (line 13) instead of Java’s `void`. It is best practice to annotate all return types for public methods. Where a method is not exposed and its return type is obvious, the type annotation can be omitted. Similarly, type annotations for fields are optional and can be omitted in many cases because of Scala’s local type inference [66].

Rather than using conventional for-loops, Scala uses ”for comprehension” which iterates items of a collection without explicitly indexing it, as shown in line 15. This is rooted in Scala’s generic collection framework, first introduced in 2010 with the release of Scala 2.8.

One important property is that each of the three collection categories (Sequence, Set, Map) extends the `Iterable` trait which provides the ”for-each” functionality used by for-comprehension. The arrow notion (`arg <- args`) in the example is syntactic sugar to shorten the underlying ”for-each” expression.

Despite these syntactical differences, Scala and Java have a high level of interoperability: Java methods can be used directly from Scala and vice versa by calling Scala’s function from Java [63][65].

### 4.2.1 Pattern matching

Scala has a built-in general pattern matching mechanism. It allows matching any kind of object with a first-match policy. Unlike Java’s Switch statement, pattern matching in Scala is complete and deterministic by default. The example in listing 4.2 illustrates a simple function that matches a given integer value to a corresponding String.

---

6http://www.scala-lang.org/node/120
Chapter 4. Scala

```scala
object MatchTest1 extends App {
  def matchTest(x: Int): String = x match {
    case 1 => "one"
    case 2 => "two"
    case _ => "many"
  }
  println(matchTest(3))
}
```

Listing 4.2: Pattern matching in Scala

A pattern in Scala is constructed from one of the following elements:

- Variables
- Variable binding patterns
- Type patterns
- Constant literals
- Named constants
- Enumerations
- Constructor patterns

Variables, as used in the example above, match any value, and bind the variable name to the value. Variable binding patterns are of the form `x@p` where `x` is a variable and `p` is a pattern. The pattern matches values and, in addition, binds the variable `x` to the matched value. Type patterns, for instance, `x : int`, matches all values of the given type and binds the variable name to the value.

Constant literals such as 1 or abc only match the referenced value. Similarly, named constants refer to immutable values, and thus only match values referring to. Enumerations are implemented as named constants in Scala, which means the same rule is applied as defined for named constants. Constructor patterns of the form `C(p1, ..., pn)`, where `C` is a case class (see 4.3.6) and `p1, ..., pn` are patterns. Such a pattern matches all instances of class `C` which were built from values `v1, ..., vn` matching the patterns `p1, ..., pn` (3 in [67]) (7.2 in [63]) (8 in [65]).
Pattern matching is often used in combination with case classes (see 4.3.6) in order to traverse class hierarchies efficiently. The example in listing 4.3 models a minimal untyped lambda calculus [68]. All three cases classes share the same super type `Term`, pattern matching can be done recursively. The `printTerm` function is of type `Unit`; it takes any term and prints it formatted to the standard console.

Case `Fun` and `App` define further a recursive call to the function in order to print nested terms correctly formatted. The return type of a pattern matching function can be arbitrary; if a particular value matches the pattern, the return type may be Boolean [67][63](8 in [65]).

```scala
abstract class Term

case class Var(name: String) extends Term

case class Fun(arg: String, body: Term) extends Term

case class App(f: Term, v: Term) extends Term

object TermTest extends App {
  def printTerm(term: Term) {
    term match {
      case Var(n) => print(n)
      case Fun(x, b) =>
        print(“ˆ” + x + “.”) printTerm(b)
      case App(f, v) =>
        Console.print(“( ”)
        printTerm(f) print(“ ”) printTerm(v)
        print(“)”)
    }
  }

  val t = Fun("x", Fun("y", App(Var("x"), Var("y")))))
  printTerm(t)
}

Listing 4.3: Pattern matching on case classes
```

4.2.2 Extractor objects

Extractor objects are used for implicit extraction of a value encapsulated in a data structure. By convention two methods, `apply` and `unapply` are defined in an object. The `apply` method, by convention, defines a so called `injector` as a method used to "inject" a value into the data structure. Conversely, `unapply` defines an `extractor` that extracts the value from the data structure. Any potential conversion is defined in an extractor to ensure the extracted type matches the one defined as parameter of the `apply` method. The major difference to similar concepts like constructors or setter and getters is that injectors and extractors are defined by a convention and are frequently used implicitly.

The example object in listing 4.4 illustrates the usage of an extractor for the object Twice. This object defines an apply function, which provides a new way to write integers: Twice(x) is now an alias for x * 2. The `apply` method emulates a constructor and defines the arithmetic operation on the passed integer. Because of the convention of using `apply` for injectors, the compiler inserts the call to apply implicitly. This allows the writing of Twice(21) which expands to Twice.apply(21). The `unapply` method in Twice reverses the construction in a pattern match. It tests its integer argument z. If z is even, it returns Some(z/2). If it is odd, it returns None. `Some` is a language construct in Scala indicating that it may or may not contain a value and thus mitigates run-time error handling to the client. Type `None`, on the other hand clearly states no return value.

```
1 object Twice {
2     def apply(x : Int) = x * 2
3     def unapply(z : Int) = if (z%2==0) Some(z/2) else None
4 }
```

Listing 4.4: Extractor in Scala

In this example, apply is called an injection because it takes an argument and yields an element of a given type. unapply is called an extraction, because it extracts parts of the given type. Injections and extractions are often grouped together in one object, because then one can use the objects name for both a constructor and a pattern extraction. Unlike case-classes, extractors can be used to hide data representations.
trait Complex

case class Cart(re : double, im : double) extends Complex

object Polar {
  def apply(mod: double, arg: double): Complex = {
    new Cart(mod * Math.cos(arg), mod * Math.sin(arg))
  }

  def unapply(z : Complex): Option[(double, double)] = z match {
    case Cart(re, im) =>
      val at = atan(im / re)
      Some(sqrt(re * re + im * im),
      if (re < 0) at + Pi else if (im < 0) at + Pi * 2 else at)
  }
}

Listing 4.5: Data encapsulation using extractors

For instance, complex numbers can be represented either as Cartesian coordinates or as Polar coordinates. The example from listing 4.5 shows an implementation for both cases using both, extractors and pattern matching for Polar coordinates.

The important aspect of using extractors is a clear encapsulation of data representations. The client uniformly uses Polar(X,Z) without any knowledge of the internal representation which simplifies code maintenance; this allows changes at any time while preserving the interface exposed to the client. Also, extractors allow easy extensions for new variants and new patterns, since patterns are resolved to user-defined methods (4 in [67]).

There are three important aspects that make extractors a useful cornerstone for Scala’s powerful expressiveness. First of all, a case class (see section 4.3.6) can be conceptualized as syntactic shorthand for a normal class with an injector/extractor object, which is implemented in a particularly efficient way. Second, correctly used, one can switch between case classes and extractors without affecting pattern-matching client code. Third, the typecase construct also plays an important role in the presence of implicits and type views (see 4.5) where implicit matching on type patterns is used for higher-level constructs defined through implicit conversions (7 in [67]).
4.3 Unified object model

Scala uses a pure object-oriented model similar to SmallTalk, which means, every value is an object; methods and functions are values and thus objects. Unlike Java, Scala unifies references and values in its object hierarchy as shown in illustration 4.1.

![Scala’s unified object model](image)

**Figure 4.1**: Scala’s unified object model

Every class in Scala inherits from the root element, Scala.Any, which specializes in two sub-classes "AnyRef" for references and "AnyVal" for values. Value classes, for example, Int, Double or Short, map to the corresponding classes in Java, but unlike Java, Scala does not have primitive types. Instead, Scala’s compiler determines if a field can be re-written to a primitive value. This is always the case if only primitive operations such as addition or multiplication are used. However, if “non-primitive” methods are used, as for instance, those corresponding to the methods defined in `Java.lang.Integer` then the compiler cannot transform such fields into corresponding primitives and uses the object representation instead.
4.3.1 Collections in Scala

Scala follows the concept of unified collections, which means there is just one syntax used for creating different kinds of collections; likewise, accessing values of many different types is done by using just one kind of syntax. For example, as shown in listing 4.6 creating an array, a map, a range or a set follows the same syntax rules. Also, as shown in the same example, assigning all different kinds of values to a set follows the same syntax\(^8\), regardless of the type [63][65]. Furthermore, Scala distinguishes between mutable and immutable collections, which mean a mutable collection can be updated or extended in place.

In contrast, immutable collections never change after creation. There are operations that simulate additions, removals, or updates, but those operations will in each case return a new collection and leave the old collection unchanged; this ensures that accessing the same collection value repeatedly at different points in time will always yield a collection with the same elements\(^9\).

```scala
val vec = Vector(1, 2, 3)
val arr = Array(1, 2, 3)
val lst = List(1, 2, 3)
val rge = Range(1, 30) // range 1 to 30
val map = Map("a" -> 1, "b" -> 2, "c" -> 3)
val set = HashSet("a" -> 1, "b" -> 2, "c" -> 3)

object UnifiedTypes {
  import scala.collection.mutable
  def main(args: Array[String]) {
    val set = new mutable.HashSet[Any]
    set += "This is a string"  // add a string
    set += 732  // add a number
    set += main  // add a function
  }
}
```

Listing 4.6: Scala’s unified collection

By default, Scala always uses immutable collections unless explicitly declared otherwise. In order to use any mutable collection, the corresponding package needs to be imported first. Also it is best practice\(^{10}\) to use the mutable prefix while declaring (line 5 in listing 4.6) to mark the implication of non-thread safety.

---

\(^8\)6.12.4 in [65]


\(^{10}\)http://twitter.github.com/effectivescala/
4.3.2 Null references

Unlike Java which has only one "null", Scala has five different kinds representing empty values or references. Table 4.1 shows an overview of all different null values and their corresponding meaning. Beginning with Null, which is a subtype of all reference types, its only instance is the null reference. Since Null is not a subtype of AnyVal, null is not a member of any value type. For instance, it is not possible to assign null to a variable of type Int. Instead, only integer values or values of a contravariant type such as Short, Char or Byte can be assigned. Assigning a value of contravariant type will be converted implicitly to an integer value (6.3 in [65]).

<table>
<thead>
<tr>
<th>Value</th>
<th>Type</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Null</td>
<td>Trait</td>
<td>N/A</td>
</tr>
<tr>
<td>null</td>
<td>instance of Null</td>
<td>null reference, like null in Java</td>
</tr>
<tr>
<td>Nothing</td>
<td>Trait</td>
<td>Empty implementation: ???</td>
</tr>
<tr>
<td>Nil</td>
<td>Type</td>
<td>defines empty list (size zero)</td>
</tr>
<tr>
<td>None</td>
<td>Type</td>
<td>no return from an option</td>
</tr>
<tr>
<td>Unit</td>
<td>Type</td>
<td>no return value from a method</td>
</tr>
</tbody>
</table>

Table 4.1: Different null references in Scala

Nothing is, together with Null, at the bottom of Scala’s type hierarchy as shown in illustration 4.1. Trait Nothing has no instances, but extends class Any which is the root type of the entire Scala type system, and consequently Nothing is a subtype of AnyVal and AnyRef. The implication is that Nothing is a subtype of everything in Scala; for instance, a String, Int or any custom class written in Scala have implicitly Nothing as shared supertype. This mechanism allows a proper definition of an empty list and empty return type.

Nil is an object that extends List[Nothing], for instance type "Nil" will be inferred as List[Nothing], which represents the empty list in Scala. Because lists are covariant in Scala, this makes scala.collection.immutable.Nil an instance of List[T], for any element of type T. Another important alias is the triple question mark ??? which is an alias for type Nothing indicating an empty implementation. For instance, a function def test = ??? will be inferred as type Nothing and compiles even though the function does not have a defined body.
None is a return type for methods that have no return value. This solves a particular situation where a method may or may not return a value. In Java, this can be done either by returning null and letting the caller perform a null check or by using the try-catch block to return different values depending on the success of the checked method.

Scala defines an Option that has exactly two subclasses; class Some and object None. Using Option as return type allows proper runtime handling of a method, that may or may not, return a value. Listing 4.7 shows a function that converts only positive integers to string.

```scala
1  def getStrin(g num: Int): Option[String] = {
2      if ( num >= 0 ) Some("A positive number!")
3      else None
4  }
5  println(getString(23).getOrElse("No number"))
```

Listing 4.7: Scala option example

The purpose of an Option[T] return type is to tell callers that the method might return a T in the form of a Some[T], or it might return None. This means, the caller handles the "None" case through the "getOrElse" method which allows specific error handling, depending on the actual case.

Unit is the return type of methods that never returns a value at all, just like void in Java. Unlike Option which may or may not return a value, Unit is used if it is clear that a method never return anything. Table 4.1 provides a brief summary for each of the different kinds of null references in Scala.
4.3.3 Object definitions

Unlike Java, Scala has no "static" modifier but uses singleton objects instead. The keyword "object" defines a single object (or: module) of a new class which may extend one or more modules using the syntax:

"object o extends m1 with m2 with... mn"

Extending another module can be omitted but the default "extend scala.AnyRef" is implicitly added to the object definition. The object body can be omitted, if required.

4.3.4 Companion object

Classes in Scala do not have static members. Instead, an equivalent effect can be achieved by defining a companion object for a class. A companion object is an object with the same name as a class or trait, and nested in the same file as the associated trait or class. Furthermore, a companion object inherits all members implicitly, thus has full access to all methods and fields including private ones.

For security reasons, a companion object exposes only public and protected members while keeping private ones only accessible within the object. Companion objects are commonly used to define implicit conversions for a class by using extractors (see 4.2.2). The example in listing 4.8 uses an abstract class Box as shared super type for several implementations for different types such as String, Boolean or Double.

In order to allow a unified access to all different kinds of Boxes, a companion object is defined that uses the injector principle (see 4.2.2) by overriding apply multiple times, one for each sub-type of Box. By creating a concrete instance of Box, an implicit pattern matching on the parameter type is performed first in order to determine which apply to call. Once a corresponding apply method is found, the expression gets re-written towards a reference to the specific apply method.

For instance, the expression val b = Box(true) gets re-written to Box.apply(true) which then, if called, references to the constructor of BooleanBox. In case an unknown type is given as an argument, an error is thrown stating no suitable apply alternative is available.
However, this is only the case if all possible implicit conversion fails. For instance passing Float or Long values to Box is valid because there are implicit conversions to type Double, which means parameters will be converted automatically. Section 4.5 provide more details about implicits in Scala.

```scala
abstract class Box

class StringBox(s: String) extends Box
class BooleanBox(b: Boolean) extends Box
class DoubleBox(d: Double) extends Box

object Box {
  def apply(s: String) = new StringBox(s)
  def apply(b: Boolean) = new BooleanBox(b)
  def apply(d: Double) = new DoubleBox(d)
}
```

Listing 4.8: Scala companion object

The double usage of the same name for the class (or trait) and the companion object is valid, since the class definition defines its name in the type name-space, whereas the object definition defines its name in the term name-space. The rationale behind companion object is twofold. The first reason is to provide static access to a class in order to ensure interoperability with Java. In particular, the Scala compiler generates companion objects automatically whenever it finds Java classes containing static members. The second reason is to provide a convenient way to incorporate factory methods or implicit conversations into Scala classes (5.4 in [65]).

4.3.5 Classes

There are two different kinds of classes in Scala: The first one is the commonly known class as a template to create objects from. Unlike traits, classes are allowed to have constructor parameters and unlike Java, a default constructor does not need to be defined, as a matching default constructor will be generated by the compiler. Constructor overloading is possible, but requires a slightly different syntax by using ”self” reference; for instance, the class list class in listing 4.9 has three constructors.
The first one is the empty default constructor generated by the compiler. The second takes a new head as argument and the third one creates a new list with a head and a tail.

Similar to Java, a class can be abstract so that no instances can be created. Classes are also allowed to inherit from multiple other modules (classes or traits) following the same inheritance syntax as objects and traits.

```scala
class LinkedList[A] () {
  var head = .
  var tail = null
  def isEmpty = tail != null
  def this (head: A) = {this(); this.head = head }
  def this (head: A, tail: List[A]) = {this(head); this.tail = tail }
}
```

Listing 4.9: Scala constructor overloading

Access protection is handled by either declaring a class private or protected. A private class may only be accessible from its companion object, which allows exposing one public interface while keeping the entire implementation private. Protected classes only share its (protected) members with classes that extend them but do not escape the scope of the extending class. A class may be ”sealed” which means any other class extending a sealed class must be located in the same source file (5.3 in [65]).

### 4.3.6 Case classes

The second kind of Scala class is called a ”case class” which is defined by using the ”case class” keyword followed by a class name and elements in brackets. Case classes do not take type parameters and may omit the body implementation. All elements of a case class are immutable by default and publicly accessible. A default implementation of equals, hashCode and ”toString” for all immutable fields defined in a case class are generated automatically by the compiler. The most common use case for case classes is the replacement for immutable Java Beans, as case classes are immutable and serializable by default (5.3.2 in [65]).
sealed abstract class Message

case class Score(caseId: Int, score: Int) extends Message

case class Similarity(sim: BigDecimal) extends Message

case class Result(simCase: Case) extends Message

object MessageProcessor extends App{
  def processMessage(msg: Message) = msg match{
    case Similarity(n) => processSim(n)
    case Score(x, b)    => processScore(x, b)
    case Result(r)      => processResult(r)
  }
}

Listing 4.10: Scala case class

A case class that extends any abstract class can be used for pattern matching and thus allow simple and thread safe control flow on data structures as illustrated in listing 4.10 (5.3.2 in [65]).

4.3.7 Traits

A trait is a class that defines members but is meant to be added to some other class as a mixin [69]. Unlike normal classes, traits cannot have constructor parameters. Since Scala does not have traditional Interfaces as Java, traits are used instead. Allowing traits to be partially implemented means defining default implementations for some methods for re-using them but may leave others without undefined in order to enforce custom implementations. Traits can inherit from other traits or from classes. Traits can be used in all contexts but unlike classes or objects, only traits can be used as mixins.

In general, interface minimalism is advocated\(^{11}\) as it leads to greater orthogonality and cleaner modularization. For instance, Java’s "java.lang.System" contains input, output and error handling altogether in one abstract class. In Scala, using interface minimalism would lead to a design having three separate traits as interfaces, each defining exactly one aspect which can then be mixed together as required; for instance by implementing an error-handled input stream [66](5.3.3 in [65]).

\(^{11}\)Object Oriented Programming in [66]
The "diamond problem" of multiple inheritance [70] is approached by using three concepts. First of all, Scala has a strict order of inheritance. For instance, several traits can be added to a class by declaring a class as:

\[
\text{Class A extends T1 with T2}
\]

Class A uses Trait T1 as first order references and T2 as second order references. Second, Scala’s class linearization (see 4.6) ensures unique references to all members in a class. Third, Scala supports constraint orthogonality to restrict the usage of traits only to certain types [63].

### 4.4 Scala type system

Scala is a statically typed language but adds a few more additions to its type system. For instance, Scala’s type system provides type inference, abstract types and members, explicit self-type annotation and existential types. Even though Scala adds several advanced concepts to its type system, it does not interfere with the usage of the language itself. For instance Scala provides declaration site variance notion for generics which makes generic Scala code, compared to Java, more readable.

#### 4.4.1 Type inference

Unlike Haskell [71] which uses global type inference enabling inferring of nearly all types, Scala only allows local type inference for some types. The reason why Scala only supports local type inference is rooted in the Hindley-Milner algorithm [72] widely used for global type inference. Essentially, Hindley-Milner type inference is not decidable in a language with sub-typing, inheritance and method overloading [73]. Overcoming these limitations is under active research [74] and may lead to improvements in the future. As Scala was designed from ground up to be functional as well as object oriented, the decision was to sacrifice global type inference in order to preserve sub-typing and inheritance which are major properties of object oriented programming.
In practice, the type of a variable can be omitted in most cases as the compiler can deduce the type from the initialization expression of the variable. For Int, Double and String, the deduction works without any hints. However, in order to distinguish Float from Double values, Float values needs to marked by using Java’s convention of appending f (or F) to the actual value. Likewise, the distinction between Long and Integer values is done by appending l (or L) to Long values, as shown in listing 4.11. In the rare case type deduction fails, the actual type needs to be declared.

Return types of methods do not need to be declared since they correspond to the type of the body which gets inferred by the compiler. It is not compulsory to specify type parameters when polymorphic methods are called or generic classes are instantiated. The Scala compiler will infer such missing type parameters from the context and from the types of the actual passed parameters. Line 12 in listing 4.11 illustrates how types of the passed parameters are used to infer the actual type parameters required to create a new instance of the generic class Pair.

In terms of limitations, the Scala compiler is not able to infer a result type for recursive functions and thus type annotation is required in this case. Also, Scala cannot infer a higher-kindred type which is a partially applied type-level function. Also, parameters types in method signatures as well as constructor arguments types are mandatory to declare.
Type information in a function's arguments list flow into the body of the function from left to right across argument lists. In a function body, types of statements flow from first to last statement deducing the final return type. This mechanism prevents Scala from inferring types for recursive functions because the last statement in a recursive function is the function itself but as the type information flow cannot go in reverse order (last to first statement) the actual type of a recursive body cannot be deduced.

Special attention is required for anonymous inner functions (lambda expression) in order to preserve the type information flow. Consider a function that takes two lists and one lambda that combines these two lists into a third one. Consequently, the function definition requires three parameter types, two for the actual lists and one for the lambda expression. Listing 4.12 shows the function definition, by using parameters only. The function definition itself is correct but once it is used, type inference on passed function arguments fails, as shown in listing 4.12.

```scala
// one parameter list; breaks type inference.
def zipWith[A, B, C](a: List[A], b: List[B], f: (A, B) => C): List[C] = {
    a zip b map f.tupled
}
scala> zipWith(List(1, 2), List(2, 3), (_ + _)) // usage in REPL
error: Unspecified value parameter f.
    zipWith(List(1, 2), List(2, 3))(_ + _)
```

Listing 4.12: Incorrect type inference on lambda expressions

The actual cause of the problem triggered by this correct function is rooted in Scala’s type inference algorithm. Essentially, type information does not flow from left to right within an argument list, only from left to right across argument lists. Applied to the function shown above, the anonymous inner function \( f : (A, B) \Rightarrow C \) fails on type inference because types \( A \) and \( B \) are unknown within the function scope. Correcting\(^{12}\) this issue only requires to rewrite the lambda as partial function in a second argument which is also known as "currying"\(^{13}\). Currying is a general technique of transforming a function that takes multiple arguments in such a way that it can be called as a chain of functions, each with a single argument.

\(^{12}\)http://pchiusano.blogspot.co.nz/2011/05/making-most-of-scalas-extremely-limited.html

\(^{13}\)The name refers to Haskell Curry who is widely credited for having invented this technique \([75][76]\)
Listing 4.13 shows the previous function but with correct type inference. A few observations are noticeable. First, currying the function pushes the lambda in a separate argument list. This complies with the rule that type information flow only from left to right across argument lists. Therefore, type A and B are defined in the first argument list and consequently flow into the second parameter list and thus there is no need to annotate the type of the lambda expression. Type information of the lambda’s function parameters will flow into the body of the function and Scala is able to deduce the correct return type $List[C]$ of the function, as shown in listing 4.13.

```
1  def zipWith[A,B,C](a: List[A], b: List[B]) // first parameter list
2      (f: (A,B) => C) = // second parameter list for lambda
3      { a zip b map f.tupled } // inferred return type is : List[C]
4  scala> zipWith(List(1,2), List(2,3)) (+ _) // usage in REPL
5  res1: List[Int] = List(3, 5)
```

**Listing 4.13**: Correct type inference on lambda expressions
4.4.2 Parameterized types

Type parameters abstract over the specific type of a value, which is also known as "Generics". The covariant generic list from listing 4.14 defines three simple methods that all depend on the actual type of the list.

```scala
abstract class GenList[+T] { // covariant generic list
  def isEmpty: Boolean
  def head : T
  def tail : GenList[T]
}
```

Listing 4.14: Type parameter in Scala

Sub-typing of generic types is invariant by default to preserve type safety under inheritance. However, in order to provide more flexibility, Scala offers a compact variance annotation mechanism to control the sub-typing behaviour of generic types as described in the next section.

4.4.3 Variance annotations

Scala allows variance declaration of type parameters of a class using plus or minus signs. Unlike Java, Scala uses the declaration site variance notion which shields the user from using complex type annotations while using generic classes. A plus ([+T]) in front of a type parameter name indicates that the constructor is covariant. Likewise, a minus ([-T]) indicates that it is contravariant, and no prefix ([T]) indicates that a type is invariant. In Scala, type variance annotation is only allowed for immutable types, mutable ones are only allowed to be invariant.

Type invariant collections follow the rule that Col[T] is only a subtype of Col[S] if and only if S = T. This particular rule enforces reliable type checks but also implies certain restrictions. For instance, the container classes in listing 4.15 are both illegal because of using mutable parameters.

In Scala, a function type requires arguments to be contravariant (-T) and the return type to be covariant (+R). From this perspective, the first error is obvious as a covariant type (+A) is used in a position where a contravariant type is expected. Also, as mutable types can only be invariant, the covariant return value is not given in this case.
The second error is less obvious because it satisfies the rule of contravariant types (+A) as argument but the same type is used as return value which must be covariant.

```
1 class ContainerCovariant[+A](var value: A)  // Error
2 class ContainerContravariant[−A](var value: A) // Error
```

Listing 4.15: Incorrect type variance

Preventing variance related issues, as shown in listing 4.16, requires some careful considerations. Using `val` to make a return type immutable resolves the issue caused by invariance of mutable types. Example one in listing 4.16 illustrates the correct variance annotation for immutable parameter types.

However, if mutable parameter types are required, there are two possible approaches. The first one is by using invariant type parameters [T] which essentially removes any variance from the signature. As this is often undesired, the second approach for ensuring covariance of the return value is by introducing a second covariant type (V) as shown in solution three in listing 4.16. Solution three uses a type bound on the newly introduced type V in order to fulfil the covariance criterion of the return type while preserving contravariance of type parameter T.

```
1 class ContainerCovariant[+T](val value : T)     // (1) correct
2 class ContainerContravariant[T](var value:T)    // (2) correct
3 class ContainerContravariant[−T, V : T](var value: V)// (3) correct
```

Listing 4.16: Correct type variance
4.4.4 Type bounds

In Scala, type parameters as well as abstract types, may be constrained by a type bound which limits the concrete values of type variables. An upper type bound \( T <: A \) declares that type variable \( T \) refers to a subtype of type \( A \). Likewise, a lower type bound declares a type to be a super-type of another type. The term \( T >: A \) expresses that type \( T \) refers to a super-type of type \( A \).

Using the example from listing 4.17, the trait Similar represents a shared super type for any class, trait or object that overrides the function \( isSimilar \); for instance, the case class MyInt in the listing. Using the type Similar as upper bound to type \( T \) in the function \( isSimilar \) allows to apply the function to many different classes as long as the type bound is satisfied.

```scala
trait Similar {
  def isSimilar(x: Any): Boolean
}
case class MyInt(x: Int) extends Similar {
  override def isSimilar(m: Any): Boolean = {...
}
object UpperBoundTest extends App {
  def findSimilar[T <: Similar](e: T, xs: List[T]): Boolean = {...
}
```

Listing 4.17: Upper type bound in Scala

Lower type bound on the other hand defines the super-type for a type \( T \). The example list in listing 4.18 illustrates a lower type bound on a prepend function. The lower bound notion allows prepending an object of a super-type to an existing list which results in a list of this super-type. This relaxes the actual type safety in the sense that even if a type definition exists, super types can be uses as well and thus it is not assured during run-time that objects of the initial type are actually used.

Looking at the usage example in listing 4.18, the first instance of the lower bounded list is of type ListNode[Null][Null](1) which allows storing of instances of type Null. As any collection or object inhering null by default, this means virtually anything can be added to the list. Adding a String (example 2) to the empty list changes the type to the shared super-type, which is String in this case, as String has null inherited by default because of Scala’s unified object model.
Chapter 4. Scala

```scala
1  case class ListNode[T](h: T, t: ListNode[T]) {
2    def head: T = h
3    def tail: ListNode[T] = t
4    def prepend[U >: T](elem: U) : ListNode[U] = ListNode(elem, this)
5  }
6  object LowerBoundTest extends App {
7    val empty : ListNode[Null] = ListNode(null, null) // (1)
8    val strList : ListNode[String] = empty.prepend("hello world") // (2)
9    val anyList : ListNode[Any] = strList.prepend(12345) // (3)
10  }
```

Listing 4.18: Lower type bound in Scala

However, as example (3) shows, adding an integer to a StringList changes the type to [Any] which is the only super-type shared between Scala’s value classes (all numeric types and Boolean) and reference classes such as String and object [63][6](3.5 in [65]). Even though lower bounded collections are very flexible in terms of objects that can be stored in, it comes at the price of reduced runtime type safety. Using parametrized types with variance annotations is usually the preferred way for collections in order to express variance under inheritance explicitly.

4.4.5 Abstract type members

Abstract type members are types whose identity is not precisely known in a defining abstract class. In order to reveal more information about possible concrete instantiations of the abstract type itself, an abstract type member must be overridden with a concrete type definition. Abstract type members are used to abstract over a specific type of a component which allows using polymorphism as well as hiding internal representation as only the abstract type is visible. Also, using abstract type members is useful to specify types shared in an inheritance hierarchy (Chpt. 2 in [77]) (5.2 in [63]) [6].

The example in listing 4.19 defines the abstract class `Buffer` which contains an abstract type member as well as an abstract value of the same type as the abstract type member. Class `SeqBuffer` overrides the inherited abstract type `t` with a tighter bound of type `T` has to be a sub-type of `Seq[U]` which means the newly introduced abstract type `U` determines the upper-bound of the Sequence `T` is bound to.
Abstract class `IntSeqBuffer` overrides type `U` with concrete type `Int` which now bounds type `t` to `Seq[Int]` within the context of abstract class `IntSeqBuffer`. This means any concrete implementation of `IntSeqBuffer` must obey this type bound.

**Inner classes and abstract types**

Traits or classes with abstract type members are often used in combination with anonymous class instantiations which overrides the actual type that should be used in the instance. The example in listing 4.20 uses an anonymous inner class to create a new instance of `IntSeqBuffer` which enforces the type bound of type `Seq[Int]`.

The anonymous class overrides type `U` as `List[U]` because `List` is a sub-type of `Seq` but it does not override `U` as type `U` which already is defined in `IntSeqBuffer`. Next, the anonymous class overrides the abstract value `element` inherited from class `Buffer` and must be of type `T` which has been defined as `List[Int]` in the context of the anonymous class.
4.4.6 Self type annotation

Self-type annotations allow the attachment of any defined implementation to another (abstract) type which will be implemented in future. To do so, Scala supports a mechanism for specifying the type of this explicitly. The example in listing 4.21 uses the abstract type Node which is upper-bounded to BaseNode to express that any implementation needs to support a `connectWith` method. The class `BaseNode` is defined as "`class BaseNode requires Node`" which re-defines this to type Node. Without this explicit self-type annotation, this would refer to its own class type which means the expression "`new Edge(this, n)`" will fail to compile because this is of self-type Edge but not a sub-type of BaseNode, as required. The self-type annotation is an elegant way to resolve this issue.

```scala
abstract class Graph {
  type Node <: BaseNode
}

class BaseNode requires Node {
  def connectWith(n: Node): Edge = new Edge(this, n)
}

class Edge(from: Node, to: Node) {
  def source() = from
  def target() = to
}
```

**Listing 4.21: Self-type reference in Scala**

The soundness of self-types is guaranteed for two reasons. First, the self-type of a class must be a subtype of the self-type of all its base classes. Second, when instantiating a class in a new expression, the compiler checks that the self-type of the class is a supertype of the type of the object being created. Because of its flexibility and sound type correctness, self-type annotation is the key construct for lifting static systems to component-based system by using the cake pattern (see 4.8 [6](6.5 in [65])).
4.4.7 Structural typing

Structural typing is used to express type requirements by an interface structure instead of a concrete type. Essentially, structural typing resolves the issue of comprehensive type redefinition which means an implementation of a structural type may have properties not previously defined. This differs from abstract type members (see 4.4.5) where specific types can only overwrite what an abstract type has already defined.

However, because of type erasures, pattern matching cannot be performed on types which mean multiple interface implementations with non-uniform type properties cannot be detected during run-time. Essentially the problem is that several implementations of a shared super type may or may not have certain properties but a client may require some of them.

Dynamic typed languages, such as Ruby, usually perform “Duck Typing” in order to determine if an assumption is true. Duck typing means if an unknown class has a method ”quack”, it is believed that class is of type Duck\(^{14}\). Assuming a test method for birds that checks if the method fly exists, the implied problem of uncertain types becomes evident if a FlyingFish (which has the fly method) is checked, but passes the test, even though a fish is not of type bird.

Instead, Scala uses Structural Typing that defines the expected structure of a type in order to enforce static type checks during compile time. For instance, if a bird is expected and the methods fly is assumed, a structural type **FlyBird** defines exactly this specification and consequently any bird without a fly method will cause a compiler error (3.2.7 in \cite{65}). The example\(^{15}\) in listing 4.22 illustrates the usage of structural typing in three steps.

First, the required classes (or traits) need to be defined. In listing 4.40 three custom classes (**Cat**, **Dog**, **Bird**) are defined by extending abstract class **Animal**. Second, specific structural types need to be defined. In listing 4.40 two custom types are used; each type defines a different requirement depending on properties that must be present in the implementing class. These properties can be abstract type members. Third, the client uses the structural type in its method which means a careful type specification decouples the client entirely from specific implementations.

\(^{14}\)http://rubylearning.com/satishtalim/duck_typing.html
\(^{15}\)http://en.wikibooks.org/wiki/Scala/Structural_Typing
// 1) classes
abstract class Animal

class Cat extends Animal { def walk(): String = "Cat walking" }

class Dog extends Animal { def walk(): String = "Dog walking" }

class Bird extends Animal { def fly(): String = "Bird flying" }

object TestClient {

// 2) defining structural type alias to the desired class

type WalkAnimal = Animal{ def walk(): String }
type FlyBird = Bird{ def fly() }

// 3) use structural type in signature

def walk(animal: WalkAnimal){ println(animal.walk) }
def fly(bird: FlyBird) { println(bird.fly) }

def main(args: Array[String]) {

    fly(new Bird)
    walk(new Dog)
    walk(new Cat)
    walk(new Bird) // Error: Bird does not have method walk!
    fly(new Cat)   // Error: Cat is not of type Bird!

}

Listing 4.22: Structural types in Scala

Unlike duke typing, it is impossible to pass a flying frog to a method that expects a flying bird. The main method in listing 4.22 illustrates three examples of correct usage of structural typed methods as well as two incorrect scenarios that will trigger a compiler error. Even if the passed type is correct, the compile process still fails if the properties expected for a structural type are not present. For instance, passing a bird to the walk method is legal because a bird is a sub-type of Animal but as bird does not have the walk method, an error occurs.

Combining structural types with self-type annotation (see 4.4.6) offers a mechanism to perform dependency validation during object creation. This resolves the issue of incorrect component usage as all requirements are checked by the compiler and thus incorrect configures component cannot occur any more. More precisely, a self-type annotation defines the actual scope of ”this” by requiring the presence of a particular type.
Structural types on the other hand further specify this type by defining a required method, signature and even the required return type. If a valid type is passed but the expected method has a different signature the compiler verification fails. The example\textsuperscript{16} in listing 4.23 illustrates the principle of structural self-type references. The class BusinessLogic specifies its self-type to a log method taking a string as argument. Important observation is that there is no class reference which means any class defining such a log method would be valid. The two traits StandardLogger and IllegalLogger both define a log method, but only the standard logger has the right signature of the log method. While mixing in any of the loggers, the compiler verifies if the structural self-type requirements are met. If this is the case, the instance is valid, as the StandardLogger example demonstrates.

\begin{lstlisting}[style=Scala]
class BusinessLogic {
    this: {def log(s: String) => log("Starting business logic")
}

trait StandardLogger{ def log(s: String) = println(s)}

trait IllegalLogger { def log = println("Wrong") }

// using REPL
scala> val b = new BusinessLogic with StandardLogger
<console>: Starting business logic

scala> val b = new BusinessLogic with IllegalLogger
<console>:9: error: illegal inheritance; self-type does not conform to BusinessLogic with AnyRef{def log(s: String): Unit}

Listing 4.23: Structural self-type reference in Scala
\end{lstlisting}

However, if the structural self-type requirements are not met by the trait mixed in, the compiler issues a warning stating that the self-type does not conform to the expected format. Consequently, the IllegalLogger can never be used with the BusinessLogic class. The benefit is obvious in this case as structural self-reference protect the client from incomplete or incorrectly implemented dependencies.

In general, structural typing is a convenient tool but as the implementation uses reflection the overall performance impact may need case wise evaluation\textsuperscript{(3.2.7 in [65])}.

\textsuperscript{16}http://engineering.zuberance.com/2012/03/02/scala-self-typing-meets-structural-types
4.4.8 Existential types

An existential type expresses that a propositional type can be satisfied by at least one member of the type implementing the existential type. Because of that, existential types have the ability to separate implementation from interface, which allows multiple implementations for a given existential type. Existentially quantified types have diametric properties compared to universally quantified types (generics). Universal types require a calling method to know the type argument, which is not the case for existential types. Also, universal types allow a called method to refer to the type argument, which again is not the case for existential types.

In practice, existential types are one way of abstracting over types. They define an "unknown" type without specifying exactly what it is, usually because the exact type knowledge is not available in current context [78](12 in [77]).

The rationale for existential types in Scala is threefold. First of all, existential types are required for interoperability with Java’s generic wildcards. Second, they are required to interpret Java’s raw types (4.8 in [79]) because they are also still in the libraries in form of non-generic types such as `java.util.List` or `Array` (10.1 in [79]). Taking a Java list as example; it is of type `java.util.List` but the type of the elements in the list is unknown because of type erasure (4.6 in [79]). This can be represented in Scala by an existential type. Finally, existential types are a way to enable reflection that is a runtime representation of types despite type erasures. Scala uses the erasure model of generics because of the need to fully interoperate with Java.\textsuperscript{17}

\textsuperscript{17}http://www.artima.com/scalazine/articles/scala_type_system.html
4.5 Implicits

Implicits in Scala have two applications. The first one is automated (implicit) conversion from one type into another one. This is called implicit conversion and provides a convenient way to add new functionality to existing libraries without modifying them directly [80]. The second application is called implicit function parameters which behave like parameters with a default value the key difference being the value comes from the surrounding scope. For instance, if a call to one or more functions passes the same value to all the invocations frequently, implicit function parameters define this value once for all invocations. The Scala keyword implicit can be used to support both cases (8 in [77]).

4.5.1 Implicit conversions

Implicit conversions have the primary intention to extend existing third party API’s without changing them directly while preserving comparability. In practice, it is often the case that the one’s own code can be extended or changed, whereas third party libraries cannot. Essentially, implicit conversations are used for defining new methods for existing classes.

Whenever one of the new methods is called, the Scala compiler calls the implicit conversion that converts the existing type into a new one that contains the newly defined method. For instance, the example (listing 4.24) shows methods for creating a reversed and capitalized string.

```
1 val name: String = "scala"
2 println(name.capitalize.reverse)
3 >alacS
```

Listing 4.24: Implicit example in Scala

Scala’s predef class that is inherited by default to any class, defines Scala’s String class as Java String but Java’s String class has neither a capitalize nor a reverse method. Instead, an implicit conversion to Scala RichString happens where these and other methods are defined.
The implicit conversion is defines as:

\[
\text{implicit def stringWrapper}(x: \text{String}) = \text{new runtime.RichString}(x)
\]

Once the Scala compiler detects an attempt to call method `capitalize` and it determines that RichString has such a method, it looks within the current scope for an implicit method that converts String to RichString. Once `stringWrapper` is found, the conversion is done automatically and the `capitalize` method is executed. The interesting aspect of implicit conversions is the preservation of the existing Java String class as well as all existing usage of it. Instead, Scala only adds new functionality to its RichString class which is never accessed directly but only through implicit conversation during compilation. However, implicits must satisfy some rules (6.26 in [65]) before the conversation can be accomplished:

1. Only methods with the implicit keyword are considered.

2. Only implicit methods in the current scope are considered, as well as implicit methods defined in the companion object of the target type.

3. No conversion will be attempted if the object or method type check fails.

4. Implicit methods are not chained to get from the available type, through intermediate types, to the target type. Only a method that takes a single available type instance and returns a target type instance will be considered.

5. No conversion is attempted if there are more than one possible conversion methods that could be applied. There must be one and only one possibility.

In some cases, neither the current scope nor the companion object of the target type can be modified in order to add implicit conversion. For instance, adding some custom functions to RichString cannot be done directly because Scala’s `Predef` and RichString cannot be modified. In this case, a custom object containing another implicit conversion from CustomString to RichString can be written and imported to accomplish this goal.(6.26 in [65])(8 in [77]) This pattern of extending third party libraries through implicit conversions is also known as `Pimp my library` pattern, first introduced by Martin Odersky [80].
4.5.2 Implicit function parameters

Implicit function parameters add support for default argument values. A method with implicit parameters can be applied to arguments just like a normal method. However, if such a method misses arguments for its implicit parameters, such arguments will be automatically provided through a default argument value within the scope. If no default value exists, an error concerning the missing implicit value is thrown. The example in listing 4.25 shows a multiplier function that takes the factor

```scala
def multiplier(i: Int)(implicit factor: Int) {
  println(i * factor)
}
implicit val factor = 2

multiplier(2) // res is 4
multiplier(8) // res is 16
multiplier(8)(4) // res is 32
```

Listing 4.25: Implicit parameters in Scala

as an implicit parameter. If the implicit value is provided within scope, any call of the multiplier function will use the same factor if not explicitly provided otherwise. However, the implicit value should never be defined within the function itself since this disables the overriding of the value which leads to unexpected behaviour (9 in [63])(6.26 in [65])(8 in [77]).

Implicits in Scala are very powerful and, at the same time, difficult to debug if used excessively. In general, using implicits requires some careful considerations such as the trade-off between implicit type conversion and introducing a new and explicit type. Also, the choice of default values for implicit function parameters should be done carefully to prevent unwanted side effects.

Wisely used implicits offers the benefit of making otherwise complex language constructs simpler to use while preserving the full power of expression. For instance, Scala’s for-comprehension shields a sophisticated concept and implementation by using implicits to define a simple syntax(8 in [77]).
4.5.3 Type views and view bounds

A view is an implicit value of function type that converts a type A to type B. A view from type A to type B is defined by an implicit value which has function type \( A \rightarrow B \) or by a method convertible to a value of that type. The arrow notion indicates that Type A can be viewed as Type B, if required. The term view conveys the sense of having a view from one type (A) to another type (B) which means type views are a way of making automatic implicit conversions available on type parameters.

In order to perform an automatic implicit conversion, type views are essentially implicit conversions being passed implicitly. As type views are implicit conversions (which are in Scala just a Function value) therefore a view can be passed as a parameter. Since type conversion is done automatically, the caller of the function is not required to explicitly pass the parameters which means those parameter are also implicit parameters. A type view is applied in two different cases:

1. When a type A is used in a context where another type B is expected, and there is a view in scope that can convert A to B. In this case, Type A will be converted automatically to type B.

2. When a non-existent member \( m \) of a type A is called but not present in type A, but only in Type B. If there is a view (in scope) that can convert A to a B it will do so automatically in order to make that member \( m \) available to the callee.

An example for the first case would be the implicit conversion from a Scala type to a corresponding Java type. For example, `double2Double` converts a `scala.Double` to a `java.lang.Double`. A common example for the second case is the initialization syntax for Maps \( (x \rightarrow y) \) which triggers the invocation of `Predef.anyToArrowAssoc(x)` and then the actual initialization method gets dispatched to the one defined in the target type.

A view bound is indicated with the \(< \% \) keyword, for instance \( A < \% B \) which allows any type to be used for A if it can be converted to B using a view. This mechanism forms a predicate in the sense of stating explicitly the conversion rule that must apply before the actual conversion will be performed.

\[18\]http://www.scala-lang.org/node/130
Chapter 4. Scala

If a passed parameter type cannot be converted to type B, as the view bound requires, the compiler will issue an error. Type views and bounds together form a convenient shorthand for expressing that a type must conform to a particular target type and if so, the type will be implicitly converted before calling the actual function on the target type.(9 in [63])(7 in [65])(8 in [77]).

The example19 in listing 4.26 come in two versions. The first version uses implicit conversion to ensure that the parameter type is implicitly converted to Ordered[T] as required for the comparison. The second version uses a type view T +% Ordered[T] to express that type T must be representable as Ordered[T] in order to execute the function body.

1 // using implicit parameter conversion
2 def max[T](a: T, b: T) // 1) part
3    (implicit $ev1: Function1[T, Ordered[T]]): T = { // 2) part
4    if ($ev1(a) < b) b else a // 3) part
5  }
6 // using type view
7 def max[T +% Ordered[T]] (a: T, b: T) : T = if (a < b) b else a

Listing 4.26: Type views in Scala

To make the first version more understandable, I’ve divided a more detailed explanation of its functionality into three parts corresponding to the numbers in listing 4.26.

The first part (max[T](a: T, b: T)) defines the basic signature with a type parameter T and two parameter a and b that must be both of type T. That means, if T is, for example, of type Int, then a and b must be both of type Int as well.

The second part of the functions begins with the implicit keyword which indicates an implicit parameter that can either be passed, or if not, will be generated by the compiler. In this particular case, the latter case should be the default.

The term after the implicit keyword does two important tasks. First, it defines an environment variable $ev1 within the scope of the max function and second, it binds $ev1 variable to a nested function that attempts to apply type T to type Ordered[T] by overriding Scala’s predefined Function1 which is inherited by default to all classes.

---

19http://stackoverflow.com/questions/9530893
Function1 is a generic function that takes just one parameter and a specified return type. In the example, Ordered[T] is the expected return type of Function1. Function1 provides a default apply method that applies the type of the parameter to the type of the return value. This means, the function definition Function1[T, Ordered[T]]: T means Function one is overridden by passing T which should be applied to a Ordered[T]. As Ordered[T] is a sub-type of T, the return type is the super-type T, which is indicated as : T.

The third part of the function takes the parameter a of the max function and passes it through the $ev1 variable to the overridden Function1 in order to convert a from type T to Ordered[T]. After the conversion, the actual comparison between a and b can be done and if a is smaller than b, a will be return. The else statement does not require any type conversion any more because if the previous statement is false, a is consequently bigger than b.

The second version uses a type view which defines that type T must be conform to type Ordered[T] and if that’s the case, the conversion will be done implicitly before applying the function to the converted parameters. Compared to the first version, the signature is much more obvious and does not even contain any implicit parameter or local environment variables bound to overridden default functions.

Instead, it just uses a type bound to express the requirements for type T in order to perform the computation in the function body. Using the function from the previous example, the next listing 4.27 shows the actual usage of the max function. The type view requires that an implicit conversion is provided for type T. This is given for all value types in Scala as they extend type AnyVal which enforces the implementation of Ordered[T] (see 4.3. If two different types are given as parameter, ”higher” type is used for conversion, as listing 4.27 shows for Long and Integer.

```
// using REPL
scala> max(3 , 9)   // res: Int = 9
scala> max("A" , "B")   // res: java.lang.String = B
scala> max(11 , 0)   // res: Long = 1
scala> max(null , 0)
<console> "error: No implicit view available from Any ⇒ Ordered[Any]."
```

Listing 4.27: Usage of type views in Scala
As the last example in listing 4.27 shows, if there is no way to use a type of the parameter list directly, the shared super type of both is used to try an implicit conversion. For null and Integer, the shared super type is Any which does not provide a view that converts type Any to Ordered[Any] and thus the implicit conversion fails and causes a compile error.

The main usage of type bound is twofold. First of all, it enforces context bounds; for instance the requirement of total ordering of the type elements in the example. The second usage is making proofs about the types that are being passed. This makes it possible to let the compiler detect issues that would otherwise result in runtime exceptions. For example, the type bound of flattened function from the Scala standard library requires that only a nested Option can be flattened. The implicit conversion fails during compilation if this requirement is not met by the type of the parameter. In both cases, type bound allows a solid compile time proof in order to determine if assumed requirements are met (7.3 in [65])(8 in [77]).

### 4.5.4 Generic type constraints

Extending the concept of type bounds further, generic type constraints are used from within a type-parametrized class or trait, to further constrain one of its type parameters. Unlike type bounds which are defined on class or trait scope, generic type constraints are defined on method scope. Scala defines three generic type constraints:

- $A =:= B$ means $A$ must be exactly $B$
- $A <: B$ means $A$ must be a subtype of $B$
- $A < % < B$ means $A$ must be viewable as $B$

An instance of $A =:= B$ witnesses that the types "$A" and "$B" are equal which can be best understood as a compiler proof of type equality. If the equal check fails, a compile error is issued. To constrain any abstract type $T$ that’s in scope in a method’s argument list, "$T <: U" witnesses that "$U" is the required upper bound for "$T".

---

20[^1]

[^1]: [http://twitter.github.io/scala_school/advanced-types.html](http://twitter.github.io/scala_school/advanced-types.html)
Conversely, "L <:< T" defines L as the lower bound which narrows the allowed type for a method down to the range of "L <=< U". The constraint, "A < % < B" witnesses that "A" is a subtype of "B via implicit conversion using a type view for implicit conversion.

The difference from general type views is that "A < % < B" constraints the type of the enclosing trait or class further, to a possible lower bound, while preserving implicit conversion for that lower bound\(^{21}\).

The functionality of generic type constraints is based on testing the feasibility of a safe type cast. The exact definition of the sub-type relation is:\(^{22}\)

```scala
sealed abstract class <=:[-From, +To] extends (From => To)
```

Beginning with the class definition, sealed means the class can only be inherited within the same source file in order to prevent unintended sub-classing. The abstract modifier ensures that there is no direct instance of the class. The identifier "::<" is a symbolic name for the class. The variance notion on the type parameters essentially means, -From is allowed to vary up (contravariant) and To is allowed to vary down (covariant). The inheritance of → refers to Function1, that is Scala’s default generic function with one parameter. In this case, the notion (From → To) serves as a function type that maps the type From to type To. The implementation of this function type is an implicit conversion from type From to type To which is essentially a type cast that verifies type A conforms to type B\(^{23}\).

In practice, generic type constraints resolve the issue of using implicit conversions into sub-types for enforcing further type bounds of the type of the enclosing trait or class. The example\(^{24}\) in listing 4.28, shows a type constraint on Int without generic type constrains. The actual difficulty here, apart from the obvious complexity, is that T can be constrained only within the context of `sumint` function (3a) using an implicit parameter to verify that it is of type IntLike. To provide evidence of type conformance, an implicit conversion to a lower bounded custom sub-type is required for each operation.

\(^{21}\)http://stackoverflow.com/questions/3427345
\(^{22}\)https://github.com/scala/scala/blob/v2.10.2/src/library/scala/Predef.scala
\(^{23}\)stackoverflow.com/questions/11488554/
\(^{24}\)https://gist.github.com/retronym/229163
Chapter 4. Scala

In the example, a `sumint` function is required to express that this function is only valid on "IntLike" (1) types, which are all types based on type Int. The companion object (1a) automatically converts any Int value in an IntLike type by using implicit conversion. To provide custom operations on a list of `IntLike` types, a custom List (3) is required that defines its operations on Intlike types.

The implicit conversion in `sumint27` takes any type T and passes it to IntLike which it applies to an implicit conversion, if available, to convert T into IntLike[T]. In order to preserve the ease of programming, another implicit conversion is used to convert any list into a custom ListW (3b) so that it appears the operation would be performed on any normal list (4).

```
1 // (1)
2 trait IntLike[X] extends (X => Int)
3 object IntLike {
4   implicit val intIntLike: IntLike[Int] = new IntLike[Int] {
5     def apply(x: Int) = identity(x)
6   }
7 }
8 trait ListW[T] {
9   val l: List[T]
10   def sumint27(implicit ev: IntLike[T]): Int = l.map(ev).reduceLeft(_ + _)
11 } // (3)
12 implicit def ListW[T](list: List[T]): ListW[T] = new ListW[T] {
13   val l = list
14 }
15 // REPL usage
16 import IntLike._
17 val intList = List(1, 2, 3)
18 intList.sumint27 // (4)
```

Listing 4.28: Type without generic type constrains

The illustrated code complexity can hardly be justified for the gain of adding type constrained operations to existing libraries. However, the outlined issue was prevalent in all Scala versions prior to 2.7 which mean the function `sumint27` showcases how type constraints were done in Scala 2.7 and earlier versions. From Scala 2.8 onwards, generic type constraints were added, which allows a much compacter notion of type constraints, as shown in listing 4.29.
The important difference compared with the previous version is that the Intlike type with implicit conversion is not required any more. However, a custom list with its implicit conversion from a normal list is still required to define the new custom functions.

The custom function sumInt28 (2a) uses the generic type constrain " T <:: Int" to denote that the parameter requires a sub-type of Int. The "evidence" that the given parameter complies with this constraint is implicitly provided by the compiler.

```scala
trait ListW[T] {
  val l: List[T] // (1)
  def sumInt28(implicit ev: T <:: Int) : Int = { // (2a)
    l.map(ev).reduceLeft(_ + _)
  }
  implicit def ListW[T](list: List[T]): ListW[T] = new ListW[T] {
    val l = list
  } // REPL usage
  val intList = List(1, 2, 3)
  intList.sumInt28
}
```

**Listing 4.29:** Type with generic type constrains

One remaining issue is the provision of implicit conversions, in particular whenever a method signature uses A < % < B which requires the presence of a type view that converts A implicitly to type B. In an ongoing effort to reduce the required verbosity of implicit conversions, compiler macros and in particular implicit materializations\(^{25}\) are under active development in order to generate type safe implicit conversions. For the example in listing 4.29, using an implicit materialization macro would reduce the required code to approximately two lines, one for the function and the second one for the macro that generates the implicit conversion. This functionality is planned for release in the upcoming\(^{26}\) Scala 2.11 version.

\(^{25}\)http://docs.scala-lang.org/overviews/macros/inference.html

\(^{26}\)Scala 2.11 is planned for 2014
4.5.5 Implicit objects

An implicit object is a singleton instance of a class or trait used to perform implicit conversion. In practice, an implicit object is often defined within a companion object (see 4.3.4) in order to convert a specific type into the type represented by the companion object. Listing 4.30 shows a trait `Semiring` that defines an interface containing the operations of a semi-ring. The corresponding companion object contains an implicit object `IntSemiring` that creates an instance for Int and defines all operations for an integer semi-ring defining all corresponding operations. Implicit resolution is done at call site which means, the companion object or trait `Semiring` needs to be in scope of the caller. Implicit objects are useful in combination with implicit parameters in order to convert a parameter automatically into the required target type [65].

```
1 trait Semiring { // (1)
2     def add(x:T, y:T): T
3     def mul(x:T, y:T): T
4     def exponent(x: T, n:Int): T
5     val unitA: T
6 }
7 object Semiring { // (2)
8     implicit object IntSemiring extends Semiring[Int] { // (3)
9         def add(x: Int, y: Int): Int = x+y
10        def mul(x: Int, y: Int): Int = x*y
11        def exponent(x: Int, n:Int): Int = if(n==0) 1 else x*exponent(x, n-1)
12        val unitA: Int = 0}
13 }
```

Listing 4.30: Implicit object in Scala

Taking the algebra example a bit further, the example in listing 4.31 illustrates how polynomials are presented as a list of single terms. The defined polynomial operations convert this list of terms into semi-rings in order to evaluate each term by reusing the method defined in the semi-ring trait.

The `Term` case class (1) defines the basic data structure of single polynomial terms. The polynomial class (2) takes a list of terms as constructor parameter and defines three functions, add, mul and eval. For brevity, the implementation details are omitted.

[^27]: http://stackoverflow.com/questions/16521938
// (1)

```scala
case class Term[T](degree: Int, coeff: T) {
  def Degree: Int = return degree
  def Coeff: T = return coeff
}
```

// (2)

```scala
class Poly[T](private val terms: List[Term[T]]) {
  def add(that: Poly[T])(implicit r: Semiring[T]) = {...}
  def mul(that: Poly[T])(implicit r: Semiring[T]) = {...}
  def eval(x: T)(implicit r: Semiring[T]) = {...}
}
```

// (3)

```scala
val p1 = new Poly(Term(Term(0, 1), Term(1, 2), Term(2, 1)))
p1.eval > 3
```

**Listing 4.31: Implicit objects as implicit parameter**

Each of the function defines an implicit conversion to semi-rings of the same type as the list of terms passed so that the (omitted) implementation for each function actually re-uses the functionality defined in the previous semi-ring trait.

The important aspect of this example is that using the class constructor creates an instance of the class *Poly* which already contains all implicit conversions; thus any operation requiring a semi-ring will convert the passed polynomial automatically.

The usage example (3) defines a polynomial p1 and calls the eval function. The eval function refers to the term list passed as constructor argument to the Poly, and evaluates each term by calling add and multiply functions defined in a semi-ring. Because of the implicit conversion inside the implicit object, the actual usage is free of any conversions or type casts. Only the Poly class needs to be imported and used according to its public API which is based on List and instances of its own. Similar to implicit objects, implicit classes are also an option to define implicit conversion [65].
4.5.6 Implicit classes

An implicit class is a class marked with the implicit keyword. This keyword makes the class primary constructor available for implicit conversions when the class is in scope. An implicit class must have a primary constructor with exactly one non implicit argument in its first parameter list. It may also include an additional (implicit) parameter list. An implicit class must be defined in a scope where method definitions are allowed (not at the top level). For an implicit class to work, its name must be in scope and unambiguous, like any other implicit value or conversion. An implicit class is implicitly transformed into a class and implicit method pairing, where the implicit method mimics the constructor of the class [81].

```scala
implicit class RichInt(n: Int) extends Ordered[Int] {
  def min(m: Int): Int = if (n <= m) n else m
}
```

Listing 4.32: Implicit class in Scala

Implicit classes have the following restrictions. They must [81]:

- be defined inside of another trait/class/object.
- take only one non-implicit argument in their constructor.
- not have any method, member or object in scope with the same name as the implicit class.

The last limitation implies that an implicit class cannot be a case class because the corresponding companion object cannot be created and used. Apart from simplifying implicit conversions, there is another application for implicit classes: defining custom language constructs.

The example\(^{28}\) in listing 4.33 shows an object containing the implicit class \textit{IntWithTimes}. The constructor of the implicit class takes an integer that defines how often an expression should be executed.

\(^{28}\)http://docs.scala-lang.org/overviews/core/implicit-classes.html
Because of the compiler transformation illustrated in the previous listing 4.32, an implicit conversion from type Int to IntWithTimes will be generated by the compiler. Whenever the function times is called on type Int, the implicit conversion to IntWithTimes will be performed and the function will be executed. The function times takes an anonymous inner function (that must return Unit\(^29\)) as argument which will be executed as often as the Int value of the constructor argument. Even though the definition of the implicit class appears to be complex, the actual usage is quite simple. As the REPL example in listing 4.33 shows, the newly defined function is a very concise expression for repeating statements.

```
1 object Helpers {
  2  implicit class IntWithTimes(x: Int) {
  3    def times[A](f: => A) : Unit = {
  4      def loop(current: Int) : Unit =
  5        if (current > 0) { f loop(current - 1) }
  6      else loop(x) }
 7  }
8 // Using REPL
9 scala> import Helpers.
10 scala> 2 times println("HI")
11 scala> HI
```

Listing 4.33: Implicit class used for custom language extension in Scala

The anonymous inner function is not limited to Unit; it can be of any type. For instance, an implicit class may define a custom function ScalarProduct which calculates the product of two vectors. As the implicit class allows type conversion, a new type RichVector can be defined, which then calculates the scalar product as the sum of the corresponding elements of the vectors concurrently for performance optimization. A possible syntax could be \(A \rightarrow B\) by using Scala’s symbolic names to express the scalar product as arrow.

Furthermore, compatibility to existing types is already covered. As the base type is still Vector (which is a standard type in Scala), only the object defining the new implicit class needs to be imported in order to use the new function ScalarProduct wherever the base vector class is already used.

\(^{29}\)Void in Java
4.5.7 Value classes

Value classes are a mechanism in Scala to represent type safe date types while avoiding allocating runtime objects. Value classes are useful for efficiently representing new numeric classes or classes representing units of measure. Listing 4.34 defines a new value class defining metric unit. The important difference compared with normal classes or case classes is, that this meter class does not actually allocate any Meter instances, but only uses primitive doubles at runtime. Also, the method definitions are valid because of Scala’s symbolic names [82].

```scala
trait Printable extends Any { def print: Unit = Console.println(this) }
class Meter(val underlying: Double) extends AnyVal with Printable {
  def +(other: Meter) : Meter = new Meter(this.underlying + other.underlying)
  def -(other: Meter) : Meter = new Meter(this.underlying - other.underlying)
  def less (other: Meter): Boolean = this.underlying < other.underlying
  def more (other: Meter): Boolean = !less
  override def toString: String = underlying.toString + m
}
// using REPL:
val x = new Meter(3.4)
val y = new Meter(4.3)
val z = x + y
```

Listing 4.34: Value class in Scala

Defining a value class requires a class to extend only AnyVal, which is the shared super type of all values in Scala (see 4.3). The novelty of this approach is that custom value types can be defined in Scala. Figure 4.2 visualizes the key concept of value classes as a custom extension of AnyVal.

Figure 4.2: Value classes in Scala’s unified object model [5].
A value class must have exactly one parameter, which is marked with val and which has public accessibility. The type of that parameter (Double in the example) is called the underlying type of the value class which may not be a value class. However, the parameter type can be either another value (like Int, Short, etc) or a custom data type represented in a normal (but not value) class. For instance, a class `Person` can be used for a value class.

However, there is a reason for using existing numeric values in order to represent new ones. The type at compile time is a wrapper, but at runtime, the representation is the actual underlying type. If the underlying type is one of Scala’s build-in types inherited from `AnyVal` and certain requirements (details in next two paragraphs) are met, the compiler will optimize a value class and use the corresponding primitive type during run-type which essentially makes a value class free of any object creation overhead [5].

Also, a value class must not define concrete equals or hashCode methods because a value class is assumed to have structural equality and hash codes. Structural equality is based on a compiler generated equal method that compares the value of the underlying type to another one. The method to generate hash codes is inherited by default from the underlying type. A value class must have only one primary constructor and must be either a top level class or a member of a statically accessible object. Furthermore, the body of a value class can only define `def`, but no `val`, `lazy val`, `var`, or nested traits, classes or objects [82].

A value class can only extend universal traits and cannot be extended itself. A universal trait is a trait that extends Any, has only `defs` as members, and does no initialization. Universal traits allow basic inheritance of methods for value classes as the example in listing 4.34 illustrates with the `Printable` universal trait. Even though a value class is essentially overhead free during run-time, there are three cases that requires the actual instantiation of a value class [82]:

1. When a value class is treated as another type.
2. When a value class is assigned to an array.
3. When conducting runtime type tests, such as pattern matching.

Whenever a value class is treated as another type, an instance of the actual value class must be created.
Chapter 4. Scala

Using the value class meter from listing 4.35, let us assume a method `add` that sums the distance of two metric values. In this case, the `add` method accepts that a value of type Distance will require an actual Meter instance which will be created implicitly by the compiler. However, if the same method would take only meter as argument type, no instance will be created. Another situation where an allocation is necessary is when assigning to an array, even if it is an array of that value class. Finally, pattern matching and run-time type checks by using `isInstanceOf` will equally force instantiation of value classes [82].

```scala
1 def add(a: Distance, b: Distance): Distance = ... // requires instantiation
2 def add(a: Meter, b: Meter) = ... // uses primitives at run-time
3 val m = Meter(5.0)
4 val array = Array[Meter](m) // requires instantiation
5 m match {
6   // new Meter instantiated here
7   case m(3) => println("Matched 3")
8   case m(x) => println("Not 3")
9 }
```

Listing 4.35: Cases of instantiation of a value class

Considering the high number of constraints and limitations implied by value classes, naturally the question of the exact motivation for including value classes arises. There is one particular scenario that makes value classes particularly useful: *Inlining implicit wrappers* [82].
4.5.8 Inlined implicit wrappers

Inlining is a compiler optimization that replaces a function call with the function body so that the operations of the procedure or function are performed "in line". For very small subroutines, the overheads of handling arguments and constructing stack frames can exceed the cost of the operations within the subroutine itself. Thus, the goal is to improve execution efficiency by minimizing stack frame construction and removing the cost of the function call and return instructions [83][84].

Implicit wrappers in Scala are either type views or implicit classes (see 4.5) which are used for implicit conversions. Combining implicit classes with case classes essentially allows the definition of extension methods, while value classes remove the runtime overhead entirely. One practical example of implicit wrappers is Scala’s RichInt class that adds new functionality to Int. The example\(^\text{30}\) in listing 4.36 shows an excerpt of a defining value class that adds a \texttt{toHexString} method.

\begin{verbatim}
1  class RichInt(val self: Int) extends AnyVal {
2    def toHexString: String = java.lang.Integer.toHexString(self)
3  } // Using REPL:
4  Val x = 3.toHexString
\end{verbatim}

\textbf{Listing 4.36: Implicit wrapper in Scala}

The actual type view is not shown in the example but because of the implicit conversion performed by a type view, the actual method call to is optimised to the equivalent of a method call on a static object, rather than a method call on a newly instantiated object. Through that mechanism, the entire call on a static object can be fully inlined and optimized by the JVM. In summary, the combination of value classes and implicit conversion provides a tool for efficient and powerful tool for modular library extension [82].

\(^\text{30}\)http://docs.scala-lang.org/overviews/core/value-classes.html
4.6 Class linearization

Mixin composition is a form of multiple inheritance but unlike traditional multiple inheritance, it does from suffer on the diamond problem. Class linearization in Scala uses a direct acyclic graph (DAG) to represent a single linear order for all of the ancestors of a class in order to resolve potential ambiguities. This single linear order includes the regular superclass chain and the parent chains of all mixed in traits [63][6]. In order to ensure a correct inheritance order, the linearization must satisfy three rules31:

1. No class or trait may appear more than once in the linearization.

2. The linearization of any class must include unmodified the linearization of any class (but not trait) it extends.

3. The linearization of any class must include all classes and mixin traits in the linearization of any trait it extends, but the mixin traits need not be in the same order as they appear in the linearization of the traits being mixed in.

One consequence of these rules is that it is not possible to create a class or trait that has multiple disjoint super classes. The exclusion of disjoint super classes constrains the allowable combinations of trait and class inheritance. Considering two classes, A and B, that extend separately AnyRef and a class that C extends A with B. Since class A and B are disjoint, there is simply no way to create a single linearization that does not include the AnyRef class more than once. These linearization rules remain true regardless of the number of classes or traits in an inheritance hierarchy. While it is possible for a class to inherit from a trait, and for a trait to inherit from a class, not all traits can be used with all classes.

A trait whose superclass is AnyRef can be mixed in with any class, but if the trait’s super-class is more specific then only classes with the same super-class or an ancestor of the super-class can mix in this trait. Considering a super-class S and a trait X that extends S, only classes that have S as super-class, or any ancestor of S as super-class can mix in trait X [63][65][6].

\[31 (5.1.2 \text{in } [65])\]
All class and trait declarations in Scala are implicitly extended by adding "with ScalaObject" at the end. This adds all of Scala’s default functionality such as DBC to each new class. As Scala’s unified object model distinguishes value classes from reference classes, so does the linearization process. The linearization of value classes (Short, Int, Double, etc) follows all the same rules of:

**ValueClass, AnyValue, Any**

Using `Int` as example, the linearization is `Int, AnyValue, Any`. For references classes, class linearization is defined as the concatenation of all linearization of extended or mixed in classes to the base class. The formal notion is: Where $L_1...L_n$ denotes each separate linearization. Also, the $\mapsto$ operator denotes concatenation where elements of the right operand replace identical elements of the left operand, following the rule:

$$\mathcal{L}(C) = C, \mathcal{L}(C_n) \mapsto ... \mapsto \mathcal{L}(C_1)$$

The following algorithm\(^{32}\) is used to determine the correct class linearization:

1. Put the actual type of the instance as the first element.

2. Starting with the right-most parent type and working left, compute the linearization of each type, appending its linearization to the cumulative linearization.

3. Working from left to right, remove any type if it appears again to the right of the current position.

4. Append the standard Scala classes `ScalaObject, AnyRef` and `Any`

\(^{32}\)(5.1.2 in [65])
Applying the linearization rules to the example\textsuperscript{33} in listing 4.37 provides a profound insight into how a mixin composition is finally resolved.

```scala
1 class C1 {}
2
3 trait T1 extends C1 {}
4 trait T2 extends C1 {}
5 trait T3 extends C1 {}
6
7 class C2A extends T2 {}
8 class C2 extends C2A with T1 with T2 with T3 {}
```

Listing 4.37: class linearization in Scala

Resolving the linearization for Class C2 requires several steps. The first is to add the type of the actual instance to a list. Second, all other classes are added from right to left which means the linearization of T3 is added first, followed by T2, T1 and finally C2A. The next step is the application of the right associative concatenation operand (from left to right) to remove type if it appears again to the right of the current position. Finally, Scala’s default classes (ScalaObject, AnyRef, Any) are appended to the linearization.

<table>
<thead>
<tr>
<th>Step</th>
<th>Linearization</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>1)</td>
<td>C2</td>
<td>Type of the instance</td>
</tr>
<tr>
<td>2)</td>
<td>C2, T3, C1</td>
<td>Add linearization for T3.</td>
</tr>
<tr>
<td>3)</td>
<td>C2, T3, C1, T2, C1</td>
<td>Add linearization for T2</td>
</tr>
<tr>
<td>4)</td>
<td>C2, T3, C1, T2, C1, T1, C1</td>
<td>Add linearization for T1.</td>
</tr>
<tr>
<td>5)</td>
<td>C2, T3, C1, T2, C1, T1, C1, C2A, T2, C1</td>
<td>Add linearization for C2A.</td>
</tr>
<tr>
<td>6)</td>
<td>C2, T3, T2, T1, C2A, T2, C1</td>
<td>Remove duplicates of C1.</td>
</tr>
<tr>
<td>7)</td>
<td>C2, T3, T1, C2A, T2, C1</td>
<td>Remove duplicate T2.</td>
</tr>
<tr>
<td>8)</td>
<td>C2, T3, T1, C2A, T2, C1, S.Object, AnyRef, Any</td>
<td>Add defaults.</td>
</tr>
</tbody>
</table>

Once the class linearization is done, an instance of a class is created by calling class constructors in the order of the most general one to the most specific one. In terms of the linear order, constructors are resolved from right to left, beginning with Any, AnyRef and ScalaObject followed by calling constructors of any extended class before finally calling the original class constructor.

\textsuperscript{33}http://ofps.oreilly.com/titles/9780596155957/ScalaObjectSystem.html
4.7 Modular mixin composition

The Scala composition model is inspired by linear mixin composition [62], symmetric mixin modules [85][86] and traits [69]. Furthermore, mixin composition forms together with abstract type members (see 4.4.5) and explicit self-type annotation (see 4.4.6) the foundation for Scala’s component abstraction that allows the re-usage of components in various contexts.

Modular composition in Scala requires that only traits can be mixed in but does not allow any trait to refer to an abstract type, only specific type references are allowed. This restriction ensures static type checking for ambiguities and overriding conflicts during composition. A trait is a special form of an abstract class which does not have any value parameters for its constructor (see 4.3.7). Using modular mixin composition follows the syntax:

```
class c extends t_1 with t_2 ... t_n
```

where \( t_1 \) to \( t_n \) are components defined as traits. A common problem in object oriented design is the modelling of class hierarchies where some classes may or may not have certain properties. Using the example\(^{34}\) shown in listing 4.38, Nature gives us a good use-case for illustrating exactly this kind of problem as some birds can swim but not fly (penguins) and other only can fly but not swim (Frigate birds) and other birds can both, swim and fly (Pelicans). Likewise, there are some fish that can only swim (sharks) but other that can actually both swim and fly (flying fish). Using modular mixin composition is a convenient way of defining class behaviour in traits and combining them together depending on the actual requirements.

Following the example, three aspects need further clarification. First, the abstract classes \( \text{Animal} \), \( \text{Fish} \) and \( \text{Bird} \) are left empty on purpose to ensure pure type polymorphism and to avoid the problem of inheriting unrelated members to sub classes. Second, both traits, \( \text{Swimming} \) and \( \text{Flying} \) use self-type annotation (see 4.4.6) to the super-type \( \text{Animal} \) ensuring both traits can only be applied to any sub-class of \( \text{Animal} \) but not to any other unrelated class. Also, any sub-class using either trait may override any of the inherited members. For instance, overriding the \textit{flyMessage} member in the implementation ensures the display of the correct message in the \textit{fly()} method inherited from the same trait.

\(^{34}\)http://joelabrahamsson.com/learning-scala-part-seven-traits/
Third, combining both traits, Swimming and Flying together can be done in order to define either a swimming bird (Pelican) or a flying fish as shown in listing 4.38.

```scala
abstract class Animal
abstract class Bird extends Animal
abstract class Fish extends Animal

trait Swimming { this : Animal =>
  def swim() = println("I'm swimming")
}

trait Flying { this : Animal =>
  def flyMessage: String
  def fly() = println(flyMessage)
}

class Penguin extends Bird with Swimming

class Pelican extends Bird with Swimming with Flying {
  override def flyMessage = "I'm a slow flyer"
}

class Frigatebird extends Bird with Flying {
  override def flyMessage = "I'm an excellent flyer"
}

class Shark extends Fish with Swimming

class FlyingFish extends Fish with Swimming with Flying {
  override def flyMessage = "I'm a flying fish"
}

object Test extends App {
  val flyingBirds = List(new Pelican, new Frigatebird)
  flyingBirds.foreach(bird => bird.fly())
  val swimmingAnimals = List(new Penguin, new Shark, new FlyingFish)
  swimmingAnimals.foreach(animal => animal.swim())
}
```

Listing 4.38: Mixin composition in Scala

Following the example, adding new categories of Animals, for instance amphibians, is simple while re-using all existing traits. More precisely, three changes are required. First of all, a new abstract class amphibians needs to be defined. Second, missing aspects need to be defined as traits. Third, specific amphibians need to be defined as well. The required changes are summarized in listing 4.39. The benefit of using traits and mixin composition become obvious as new classes reuse previously defined traits in order to model the actual class behaviour.
A flying frog, for instance, is able to perform short distance gliding flights [87] but is still an amphibian, and thus can swim and dive as well as moving on land. In traditional object oriented modelling, such a special case would usually require a certain level of compromise.

For instance, Java would require either extending an abstract class by adding another inheritance level at the risk of inheriting non-related members to non-amphibian animals or implementing a set of interfaces for each class even though the code could be shared. Assuming an abstract animal class may contain generic methods such as flying, swimming, etc. shared amongst all animals. A second abstract class would then be used to define animal specific methods shared amongst all (or most) birds. Next, the abstract bird class is then further extended by a specific bird classes.

Adding an amphibian category later on exposes two shortcomings of the approach. First, as some amphibians require methods from nearly all other animals, there are only limited choices. Either some code gets duplicated in an abstract class shared amongst all amphibians or all methods are placed in just a single abstract class shared amongst all animals.

The first choice of code duplication has some issues with maintenance and the second choice of centralizing everything in one abstract class breaks object oriented design twice. First by inheriting several unrelated methods to any sub-classes and second has (naturally) a low level of cohesion as a direct consequence.
Scala on the other hand simplifies code re-usage as long as traits have a high level of cohesion which is best accomplished by defining one trait for one function. Also, it is important to note that the client code remains unchanged as long as no references to specific types of animals are used, in order to abstract over specific types so that categories of animals are described only by properties. In the example shown in listing 4.40 structural types (see 4.4.7) are used since not all bird classes have the same properties. The client code performs checks on the structural types in order to determine if certain methods are present.

```scala
trait AnimalTypes { // structural types
  type FlyBird = Bird{ def fly() }
  type SwimBird = Bird{ def swim() }
  type FlySwimBird = Bird{ def fly(): def swim() }
}
object BirdClient extends AnimalTypes {
  def flyBird(bird : FlyBird) { bird.fly }
  def swimBird(bird: SwimBird) { bird.swim() } 
  def swimFlyBird(bird : FlySwimBird){
    bird.fly
    bird.swim()
  }
  def main(args: Array[String]) {
    flyBird(new Frigatebird)
    swimBird(new Penguin)
    swimFlyBird(new Pelican)
  }
}

Listing 4.40: Structural typing in Scala
```

By using structural types, the client code is essentially free of any reference to any particular bird implementation. Furthermore, types that are defined in a trait can be shared application wide, which centralizes code maintenance and ensures all affected components are using the same type specification. Another important aspect is that structural type are decoupled from the actual data type which means if the Type class Bird is replaced by another class BigBird, the client code is unaffected as long as all expected methods are present in the new BigBird class. The next section provides more details about how Scala supports component decoupling by using the cake pattern.
4.8 Cake pattern

4.8.1 Overview

The cake pattern was first published in *Scalable Component Abstractions* [6] summarizing Martin Odersky’s experience of restructuring Scala’s compiler towards a modular component system. The cake pattern combines abstract types (see 4.4.5), self-type annotation (see 4.4.6) and mixin composition using traits (see 4.3.7) and companion objects in order to centralize component configuration and instantiation. One central aspect that separates the cake pattern from many component and dependency injection frameworks such as Spring or Guice is the fact that Scala’s components are statically checked by the compiler.

Unlike DI containers using auto-wiring (see 4.8.3) a missing dependency causes a compiler error and thus allows early detection during development. Furthermore, unlike many other dependency injection and component based approaches, Scala does not require any third party framework or DI container. Instead, the cake pattern uses only language constructs [6][88][89]. The next section provides a deeper insights into the underlying motivation of dependency problem.
4.8.2 The dependency problem

Wiring several dependencies together is a common task in software development and often causes problems related to maintainability. According to ISO 9126 [90], maintainability defines four sub-qualities

- Analyzability
- Changeability
- Stability
- Testability

These four criteria are used to evaluate commonly used concepts to handle dependencies in programming. The simplest form of wiring dependencies is to call constructors directly from the client. The positive aspect is that the code is simple to analyse. The drawback is that changes needs to be done carefully in order not to affect stability. Testability is a particular problem as components that call other constructors directly cannot be tested independently.

A basic form of dependency injection is passing dependencies to the constructor which is also known as constructor injection. In this case, the code is simple to analyse and simple to change because a new implementation can be passed without affecting the client (that uses the interface). This improves stability but makes testing more difficult as dependencies between components now need to create other, non-related, sub-dependencies first before the actual instance can be created.

As a workaround, the factory pattern is often used to centralize component creation and configuration. The factory pattern follows the concept of inversion of control which essentially reverses the dependency. Instead of depending on other components, a component depends on a factory that returns instances of the required dependencies. Through that, configuration and instantiation is maintained in the factory. (13 in [91])

Analyzeability of the code becomes simpler as all complex configurations are centralized in a factory. Changeability is simplified to a certain degree because any configuration changes are shielded from the client. Signature changes still need to be maintained in client code but correctly designed interfaces minimizes the issue.
Stability should improve as the introduction of new components does not affect other components, only the factory needs to be modified. Testability becomes an issue because a factory does not define how to swap implementations for an interface and depending on the configuration of a component, this can cause problems.

The main problem caused by factories is that all dependencies are hidden in the factory but not the component itself which minimizes the re-usability of components outside the factory context. To minimize the instantiation and configuration issues, the builder pattern (22 in [91]) can be used to transfer the dependency resolution process to a builder class that must be bundled to a component.

However, this only shifts the problem from the factory to the builder and thus does not resolve the underlying cause of inadequate dependency management. Instead, it adds another layer of abstraction which only exists for the purpose of managing dependencies of a component. In order to provide better dependency management, the dependency injection pattern is widely used. The next section provides more details about dependency injection.
4.8.3 Dependency Injection

Dependency Injection (DI) is a design pattern that attempts to decouple dependencies from each other and making it possible to change them, whether at run-time or compile-time [92]. Essentially, the core concept of DI is to separate component behaviour from dependency resolution. The dependency injection concept is defined by Martin Fowler as:

*The basic idea of the Dependency Injection is to have a separate object, an assembler, that populates a field in the [...]class with an appropriate implementation for [...] the interface... – Martin Fowler*

Martin Fowler identifies three ways in which an object can obtain a reference to an external component, according to the pattern used to provide the dependency [93]:

- **Type 1**: interface injection, in which the exported module provides an interface that its users must implement in order to get the dependencies at runtime.

- **Type 2**: setter injection, in which the dependent module exposes a setter method that the framework uses to inject the dependency.

- **Type 3**: constructor injection, in which the dependencies are provided through the class constructor.

An additional form of dependency injection is based on annotations which became popular after Martin Fowler’s initial article was published online back in 2004. Dependency injection using constructor injection removes the need for factories but in return exposes dependencies in the API signature which violates the encapsulation principle by exposing implementation details publicly. Also, DI in general, requires the client to lookup a component which actually mixes concerns since a client does not need to know how to locate a component. In order to extract this responsibility from the client, the locator pattern is usually applied to dependency injection which means not the client but a lookup services performs the component lookup [93].

---

[92] Inversion of Control Containers and the Dependency Injection pattern [93].
Because of the complexity of component wiring, configuration and lookup, dependency injection is commonly implemented either as a framework such as Spring\textsuperscript{36}, google Guice\textsuperscript{37} or part of a component container such as Pico\textsuperscript{38}. For the next paragraphs, Pico is used to illustrate an example\textsuperscript{39} how to apply these principles in practice. Let’s propose a virtual juicer that contains of a peeler that peels a fruit and passes the peeled fruit to a juicer in order to make juice. Assume further, that each component has its own interface and the juicer depends on the peeler and the peeler required fruits to perform the peeling task.

Listing 4.41 illustrates interfaces and implementations of the use case, written in standard Java code. The first observation is that the bottom level components (Apple, orange) are just plain Java Beans which do not contain any specific DI code. The second observation is that the actual business logic that performs a task is required to implement Pico’s generic \textit{Startable} interface that defines a \textit{start} and \textit{stop} method. This convention enforces the best practice of having just one component for one task which leads to a high level of cohesion. Also, implementing \textit{Startable} allows the Pico container to manage the entire life circle of a component. Third observation is, any business logic is required to define a constructor that defines all required dependencies, following the concept of constructor injection.

Adding components to the container can be done in two different ways. Either, all components are added to just one instance or different components are added to a hierarchy of containers, following the composer pattern\textsuperscript{40}. Listing 4.42 shows the actual usage of the Pico container by adding components to it and how to get a managed instance from the container.

Three observations are noticeable. First the general overhead in using Pico is considerably low and adds very little complexity. Second, a child container must be registered explicitly to a parent container otherwise the two containers cannot interact. The example shows that a parent container is passed as parameter to the child container, for instance $y = \text{new DefaultPicoContainer}(x)$; where $x$ is the top parent container and $y$ a child of $x$.

\textsuperscript{36}http://www.springsource.org/spring-framework
\textsuperscript{37}http://code.google.com/p/google-guice
\textsuperscript{38}http://picocontainer.codehaus.org/
\textsuperscript{39}http://picocontainer.codehaus.org/introduction.html
\textsuperscript{40}(26 in [91])
Chapter 4. Scala

Third, when requesting an instance of the Juicer class, Pico creates the right objects with all arguments and dependencies resolved at run-time. This functionality is also called **auto-wiring** which is a central property of all modern DI frameworks. Auto-wiring implies that object creation can only fail if required dependencies are not present in the container instance. This either happen if a component has not been added or if a hierarchy of containers is accessed from the wrong level.

```java
public interface Peelable {
    void peel();
}

public class Apple implements Peelable {
    public void peel() {
        System.out.println("Peeling an apple");
    }
}

public class Orange implements Peelable {
    public void peel() {
        System.out.println("Peeling an orange");
    }
}

public class Peeler implements Startable {
    private final Peelable peeler;

    public Peeler(Peelable peeler) {
        this.peeler = peeler;
    }

    public void start() {
        peeler.peel();
    }

    public void stop() {}
}

public class Juicer {
    private final Peelable peeler;
    private final Peeler peeler;

    public Juicer(Peelable peeler, Peeler peeler) {
        this.peeler = peeler;
        this.peeler = peeler;
    }
}
```

**Listing 4.41:** Components for DI in Java

For instance trying to create an instance of a component added to a lower level from a higher level of the same container hierarchy, the instantiation fails with a null pointer exception as the last line in listing 4.42 illustrates.
Pico shows cases the current state of the art of dependency injection which is also used in similar frameworks like Google Guice or Spring. However, there are certain limitations implied by these frameworks that are based on auto-wiring.

First, there is no static compile-time checking that dependencies are satisfied. Instead, null pointer exceptions are thrown which makes testing compositional components mandatory.

Second, auto-wiring is implemented using reflections which affects performance. Also, if a DI framework does class-path scanning, a common practice for locating components, names spaces clashes and other unexpected problems will be exposed during run-time.

Third, a more serious problem is that many DI frameworks (Spring, Pico) are not able to create instances from components that are not managed by the container, which means legacy or third party code must be wrapped into components order to be accessible by the framework [94].
Another issue, less obvious but inherently coupled with complex dependencies, is the order of component configuration needs to be handled at component creation time. Using the juicer example from listing 4.42, the problem becomes obvious in case different peelers are required for different kind of fruits. For example, an orange peeler applies to apples and oranges but a melon peeler only applies to melons. In this scenario, the specific peeler depends on the type of the passed fruit but the specific configuration of the juicer depends on a specific instance of the peeler. However, defining constraint propagation (If peeler is MelonPeeler then use melon configuration) during constructing the juicer component adds a static reference to a concrete implementations of the peeler interface which breaks both encapsulation as well as component decoupling.
4.8.4 Re-thinking dependency injection

Re-thinking dependency injection leads to three insights. First, class usage cannot be assumed to be independent from its implementation. Instead, it is common practice to write one implementation for a specific usage. Consequently, the separation of an implementation from its interface only requires to re-factor any context specific information into an abstract representation in form of an abstract type which is accessible only to relevant components. This allows the context re-definition for different component implementations without breaking the interface.

Second, component creation and configuration depends on its context but not always. In order to enable context awareness component configuration, a more sophisticated builder mechanism is required that provides a set of pre-defined configurations that can be applied in various contexts. This also prevents the issue of static references to specific instances because a component can provide different kinds of instances depending on the requested configuration.

Third, component creation and instantiation requires compile time verification. As already the (trivial) juice example in listing 4.42 illustrates, null references can be triggered at any time by just accessing the wrong level of a component container hierarchy. Unfortunately, these errors cannot be detected at compile time because auto-wiring is performed at run-time using reflection. Also, configuration constraint propagation requires: a) compile time checking in order to verify if the assumed requirements are met. b) Static references must be removed between components in order to prevent violations of the encapsulation and decoupling principles. c) Dependency injection needs to be done for each component separately in order to truly decouple components from each other.

Using the perspective of context switching, sophisticated component builders, per component dependency injection and compile-time verification, the next section introduces the cake pattern approach.
Chapter 4. Scala

4.8.5 The cake approach

The cake pattern aims to modularize component based development in Scala by building on top of three abstractions\(^{41}\) that are already part of the programming language:

1. Abstract type members. (see 4.4.5)
2. Explicit self-types. (see 4.4.6)
3. Modular mixin composition. (see 4.7)

Together, these three abstractions form the foundation for the cake pattern used for scalable component based development in Scala. Scalable means the same three principles of the cake pattern are applicable and usable in small systems as well as large scale systems.

A component consists of three parts, a public interface, one or more implementations and a companion object belonging to the component. Conveniently, all three parts of a component can be defined either in one single file or split in separate files if required. The public interface is a trait that defines public methods as well as an abstract member. An implementation consist of a trait implementing the interface using one (or more) nested private classes and overrides the abstract member with a specific instance of a nested class.

The companion object is a Scala object that defines which implementation is used for the declared abstract type representing the component. Using this approach means that any field assign to a static member of the companion object contains a new instance without explicitly creating one.

The underlying motivation is that the implementation of a component can be changed at any time in the companion object. Also, configuration of a component should be done centrally in a companion object which means, a companion object always returns a correctly configured instance of the corresponding component. This is particular important as not only configuration needs to be done but also all dependencies needs to be satisfied prior to configuration. The cake pattern enforces compiler verification of dependencies during component composition.

\(^{41}\) (1 in [6])
Component composition is based on modular mixin composition (see 4.7) to create composite components out of smaller ones. Component composition automatically associates required with provided services matching by name. For instance, if component shoe contains an abstract type member that defines a required component lace, the compiler looks that name up and if a component lace exists in scope, the wiring is done automatically.

By convention concrete type members always override abstract members matching the same name. These two mechanisms enable component services without explicit wiring. Instead, the matching is interfered and checked automatically by the compiler [6]. The cake pattern is shaped by several unique properties such as [6][88]:

- Service oriented component model
- Hierarchical components.
- Composite component encapsulation.
- Component based creation and configuration
- Component based dependency injection.

The cake pattern introduces a "Service-Oriented Component Model" which defines a components as units of computation that provide a well-defined set of services. A service means a particular computation a component either provides or requires. Furthermore components implementing services that rely on a set of required services provided by other components. Figure 4.3 illustrates the concept of a component. Required services are defined by using self-type annotations whereas provided services are implemented as nested class [6].

Hierarchical components allow users to express their component structure in a rich, tree-like fashion. This tree hierarchy serves the purpose to design and enforce design and security constraints. Design constraints are defined by using self-type annotations (see 4.4.6) in order to enable static compile checks. This enforces component access conforming to the specified design constraint and prevents any non-confirming usage [88].
Composite component encapsulation hides the details of a composite component from other components. This is accomplished by including sub-components only in the parent component implementation, but excluding them from the parent component interface. In practice, the interface trait only provides abstract signatures of method the component provides but the implementation may consist of several nested or even external classes depending on each other. Components are extensible entities which mean they can evolve by sub-classing and overriding. Furthermore, any component can be used to add new or upgrade existing services to other components [6][88].

Component creation and configuration is done for each component separately in a companion object that offers access to fully configured instances through a final member. A final member is an abstract member overridden with a specific type accessed through either a constant that returns one and the same instance for all invocations or a method that returns different instances for each invocation. The access through a constant is useful to define an instance that is used system wide while preserving its specific configuration. By doing so, it allows to switch between different implementations of the interface in case multiple implementations are present. Even further, by shifting the entire component creation process into a trait, multiple but different configurations can be created and used by simply composing the traits in different combinator objects [6][88][89].

Component based dependency injection is done by using self-type annotations to declare, resolve and check dependencies for each component. The following section provides a detailed example of how all these concepts work together.

---

42Self-type annotation are checked during compile-time
I use an user repository as example [88] in order to illustrate the cake pattern in practice. A user is defines by a case class that has two fields, a username and a password, as shown in the first line in listing 4.43. The first component consists of just two parts, one interface and one implementation. The interface UserRepositoriesComponent defines an abstract type member userRepository that belongs to the actual interface. It is important to note that abstract means, there is no specific type associated with userRepository. Next, the interface contains a nested (protected) trait UserRepository (note the upper case U) that actually defines the methods that determine a valid implementation of type userRepository. In the example, there is just one method signature (create) defined.

Second the implementation UserRepositoryComponentImpl inherits from the interface which means the abstract type member and the protected trait needs to be overridden in order to be valid implementation of that interface. The first part of the component overrides the abstract type member userRepository by a reference to a new instance of its nested class.

The second part of the component is a nested private class which is the actual implementation of the trait UserRepository that defines the structure of the abstract type userRepository. By doing so, the nested class UserRepositoryComponentImpl overrides the signature of create with a specific method.

A component is not limited to just one implementation, as the example shows, a second LoggedUserRepositoryImpl is also defined within the component. Changing the internal implementation of a component is as simple as overriding the abstract type with LoggedUserRepositoryImpl instead of the previously used UserRepositoryImpl. Through this mechanism, the internal implementations details are fully hidden from the public interface while preserving the flexibility of changing the internal representation at any time. This allows, for instance, to re-run performance tests with different implementations of the same component. In this particular case, there is no companion object required because this component neither requires any configuration nor complex instantiation.
Chapter 4. Scala

```scala
1  case class User(username: String, password: String)
2  /** Interface */
3  trait UserRepositoryComponent {
4      // Abstract type member (ATM)
5      protected val userRepository: UserRepository
6      // Corresponding structure of the ATM above
7      protected trait UserRepository { def create(user: User): Unit}
8  }
9  /** Implementation */
10  trait UserRepositoryComponentImpl extends UserRepositoryComponent {
11      // overriding abstract type member with specific (internal) implementation
12      override protected val userRepository: UserRepository = new UserRepositoryImpl
13      // First internal implementation
14      private[this] class UserRepositoryImpl extends UserRepository {
15          override def create(user: User) = println("create " + user)
16      }
17      // Second internal implementation
18      private[this] class LoggedUserRepositoryImpl extends UserRepository {
19          override def create(user: User) = log("create " + user)
20      }
21  }
22
Listing 4.43: First component for cake pattern in Scala
```

For security, abstract members and the interface are declared protected. Protected in Scala acts noticeably different from protected in Java. First of all, protected members are only accessible in sub-classes but cannot be accessed from classes within the same package. Second, if a broader access is required, the access scope must be explicitly defined by using a parameter `protected[foo]` where foo can be a class, package, object or `this` for referencing to its instance. Likewise, the private nested class is limited to the scope of `this` which means only members of the same instance are accessible (5.2 in [65]).
Adding a second component (listing 4.44) follows the same two steps, first, an interface (`UserServiceComponent`) and second an implementation (`UserServiceComponentImpl`) is required. The interface defines an abstract type member (`userService`) and a nested trait `UserService` (notice capitol U) that defines the actual structure of the abstract type.

The implementation trait contains a nested private class used to override the abstract member of the interface. However, the implementation of the second component depends on the first component which is specified by using a self-type annotation (`self : ... →`) to indicate the component can access all methods defined by `UserRepositoryComponent`. As the self-type annotation is defined in the implementation which means the interface `UserServiceComponent` is free of any internal dependencies.

```
 1 /* *** Interface */
 2 trait UserServiceComponent {
 3   protected val userService: UserService
 4
 5   protected trait UserService {
 6     def create(username: String, password: String): Unit
 7   }
 8 /* ** Implementation ***/
 9 trait UserServiceComponentImpl extends UserServiceComponent {
10   self: UserRepositoryComponent =>
11   override protected val userService: UserService = new UserServiceImpl
12
13   private[this] class UserServiceImpl extends UserService {
14     override def create(username: String, password: String) =
15       userRepository.create(User(username, password))
16 }
```

**Listing 4.44:** Second component for cake pattern in Scala

It is important to note that the self-type is switching the scope of `this` to the context of `UserRepositoryComponent` which means all defined fields become accessible. Through that, the field `userRepository` from the `UserRepositoryComponentImpl` becomes accessible and as this field refers to the actual implementation, the `create` method gets correctly dispatched.
Chapter 4. Scala

Changing the internal implementation of `UserRepositoryComponent` to the nested inner class `LoggedUserRepositoryImpl` does not affect the second (or any other) component since the type `UserRepositoryComponent` remains unchanged because of the abstract type member `userRepository` used in any implementation.

Composing components is done in three steps, as illustrated in listing 4.45 First, one trait mixes the required interfaces together in order to define the combined interface publicly exposed by composite component. Second, all required traits implementing each of the interfaces are mixed into one single trait. Third, one composite trait mixes the composite interface and the composite implementation together. Optionally, a companion object for the composite trait can (and should) be created in order to allow static access to the newly defined composite component. The last line in listing 4.45 shows a companion object belonging to the composite trait.

```
1 /** composed interface **/
2 trait ComponentRegistry extends UserServiceComponent
3     with UserRepositoryComponent
4
5 /** composed implementation */
6 trait ComponentRegistryImpl extends ComponentRegistry
7     with UserServiceComponentImpl
8     with UserRepositoryComponentImpl
9
10 /** composite trait, matches interface to implementation by name */
11 trait ComponentModule extends ComponentRegistry
12     with ComponentRegistryImpl
13
14 /** companion object, for static access **/
15 object ComponentModule extends ComponentModule
```

**Listing 4.45**: Cake component composition
As listing 4.45 already indicates, the newly defined composite component is used either by mixing the trait in or by importing the companion object for static access to its inherited members. Listing 4.46 shows both usage scenarios. The interesting aspect of using the cake pattern is the fact, that the abstract members `userService` and `userRepository` are only defined in the composite interface without any references to any implementation which means swapping implementations does not affect client code as long as the composite interface remains unchanged.

```scala
1 object Bakery extends ComponentModule{
2 2
3   def main(args: Array[String]) {
4 4
5     // static access through companion object
6     ComponentModule.userService.create("User1", "Test")
7     ComponentModule.userRepository.create(new User("User2", "Test2"))
8 8
9     // using mixed-in trait
10    userService.create("User3", "Test3")
11    userRepository.create(new User("User4", "Test4"))
12 12 }
13 }
```

**Listing 4.46: Cake component usage**

Swapping implementation is done by creating a new composed implementation trait and either replacing the previous one mixed into the composite trait or by simply creating a new composite trait with a corresponding companion object. Considering the composed interface from listing 4.47, let’s assume three implementations, a `Mock` version for testing, a `Prod` version for production and an `Exp` version for experimenting with interesting performance improvements. Let’s assume all of these versions are already implemented in separate cake components, the required composite traits and companion objects would look like the one in listing 4.47.
Even though it would be tempting to define another companion object following the builder pattern in order to centralize object creation and configuration in one place, this is not desirable in this particular case. Instead, a builder method is defined inside the actual application by using pattern matching in order to allow application context switching. The decision to place a builder method inside the application depends on the actual context. If a component needs to be configured in different contexts, then using a companion object is the best choice to centralize all configurations. In this case, the application configuration has just one context; thus the builder method is located inside the application.

Listing 4.48 illustrates the concept in three steps. First, an abstract class for the application is used to define commonly used methods like `start`. The class is abstract in the sense that it does not provide any kind of implementation for the defined methods, it only provides a signature.

Second, a companion object is defined in order to create a singleton instance of the abstract class. Also, all abstract methods need to be overridden with specific implementations. In order to enable application context switching, an enumeration is used for pattern matching in the `appBuilder` function. It checks the parameter and depending on the actual value, it returns a different instance of the host object. Even though the example shows only a pre-defined enum passed as an argument to the `appBuilder`, this value can be acquired dynamically during application run-time, for instance by a graphical user interface asking for switching the context of an application.
Third, in order to start the actual application, the newly created instance needs to be stored in an immutable field in order to access the overridden methods, such as `start`. Once these steps are done, the application can be started using the desired configuration. Switching from one instance to another means losing all temporary data because no context is defined in the example. However, to make dynamic context switching complete, only a `getContext` method needs to be implemented for each of the component modules in order to receive the context and pass it to a different instance. The implementations details are not covered as it exceeds the scope of the cake pattern but dynamic context switching is just one example of the benefits implied by using the cake pattern.

```scala
object Mode extends Enumeration {
  type Mode = Value
  val Mock, Exp, Prod = Value
}

abstract class Bakery {
  def start
}

object Bakery extends Bakery{
  import Mode.
  override def start = {println("Starting App")}
  def main(args: Array[String]) {
    val app = appBuilder(Mode.Mock)
    app.start
  }
  private[this] def appBuilder(mode : Mode) = mode match {
    case Mock => (new Bakery with ComponentModuleMock)
    case Prod => (new Bakery with ComponentModuleProd)
    case Exp => (new Bakery with ComponentRegistryExpImpl)
  }
}
```

Listing 4.48: Dynamic context switching using cake pattern components

However, the drawback of using the cake pattern is three-fold. First of all, the level of required knowledge and skills is high. The underlying language constructs such as self-types and abstract type members are fairly sophisticated and require a certain level of development skills. Good documentation is available but also demands a profound understanding of component based development.

Second, both component definition and composition is considerably verbose compared to the usual programming in Scala. Mature frameworks like spring have already reduced the amount of programming required to define and compose components to
a minimum. As the cake pattern is considerable new\textsuperscript{43} the level of maturity and usability is not yet on par with existing frameworks.

Third, debugging complex compositions can be difficult because of misleading and difficult to understand error messages related to illegal inheritance caused by explicit self-types annotations. It is often not clear what exactly triggers the compiler to fail and thus a certain level of comments on self-types are required to clarify what kind of inheritance is applicable.

Approaching the second issue of verbose coding, there are currently\textsuperscript{44} two attempts to combine component based dependency injections with compiler macros in order to minimize required programming. The first one is called \textit{MacWire}\textsuperscript{45} developed by Adam Warski\textsuperscript{46} with the goal to develop "Framework-less Dependency Injection with Scala Macros" \cite{warski:2005}. Essentially, \textit{MacWire} generates new instance creation code of given classes, using values in the enclosing type for constructor parameters, with the help of Scala Macros. Listing 4.49 shows an example how MacWire is used in practice. Even though MakeWire supports scopes, it does not support any of the unique properties of the cake pattern. Instead, it only replaces the functionality of DI containers such as Pico with macros.

egin{verbatim}
trait UserModule { import com.softwaremill.macwire.MacwireMacros._

lazy val theDatabaseAccess = wire[DatabaseAccess]
lazy val theSecurityFilter = wire[SecurityFilter]
lazy val theUserFinder = wire[UserFinder]
lazy val theUserStatusReader = wire[UserStatusReader]

}  \* code, generated by MacWire
*trait UserModule {
  * lazy val theDatabaseAccess = new DatabaseAccess()
  * lazy val theSecurityFilter = new SecurityFilter()
  * lazy val theUserFinder = new UserFinder(theDatabaseAccess, theSecurityFilter)
  * lazy val theUserStatusReader = new UserStatusReader(theUserFinder)
  */

Listing 4.49: Dependency injection using MacWire
\end{verbatim}

\textsuperscript{43}first published in 2005 in [6]
\textsuperscript{44}April 2013
\textsuperscript{45}https://github.com/adamw/macwire
\textsuperscript{46}http://www.warski.org/blog/
The second attempt is a set of requirements for five type macros written as a blog-post\textsuperscript{47} by John Sullivan. Essentially, he proposes to replace central language constructs such as self-types with simpler to understand macros in order to generate complex code by the compiler. So far, this concept has not been implemented. Despite the three main shortcomings, complexity, verbosity and debugging self-types, the overall benefit of the cake pattern cannot be underestimated. The benefits of the cake pattern are a clear component design, a clear separation between interfaces, implementation, component assembly and client usage. The concept of composing components without static references but with static checking during compilation enables a high level of flexibility during development. Correctly used, the cake pattern allows a component re-usage by standard (mixin) inheritance which in return reduces unnecessary code duplication. Furthermore, advanced concepts like application context switching are surprisingly simple to implement once the cake pattern is in use.

In summary, considering the complexity of the cake pattern, it is best applied to projects that are intended to grow large or for already large projects that are in need for refactoring in order to simplify complexity. For small projects, the complexity of the cake pattern may not outweigh its benefits. If code is shared amongst several different projects, the benefit of components and simplified context-switching clearly advocates the adoption of the cake pattern regardless of the actual project size.

\textsuperscript{47}http://scabl.blogspot.co.nz/2013/03/cbdi-2.html
4.9 Functions

The paradigm of functional programming is based on evaluation of (mathematical) functions and avoids state, side-effects and mutable data [95]. A state in a program is a term for specific value, stored in a memory location, at a given point in time [96]. In imperative programming, a program is described by statements changing state. In contrast, functional programming prevents changing states by using functions that avoids side effects [7]. Side-effects are:

- Reassigning a variable
- Modifying a data structure in place
- Setting a field on an object
- Throwing an exception or halting with an error
- Printing to the console or reading user input
- Reading from or writing to a file
- Drawing on the screen

Figure 4.4: Side effects in functions (1.1. in [7]).

The first three side effects changing states and thus should be generally avoided. The remaining side effects are handled in so-called "non-pure" functions. Pure functions widely considered to be side-effect free whereas non-pure have one or more the listed side-effects. Functional programming motivates implementing programs with a pure core and thin "outer layers" for handling effects. Mutable data are any data that can be changed after creation. Variables are mutable if the assigned value can be changed after creation. The opposite concept is called immutable data which cannot be changed after creation. Functional programming relies on immutable data even though Scala supports both, mutable and immutable data and data-structures [63][9](1.1. in [7]).

The next pages will just briefly cover some basic aspects of functional programming in Scala but does not provide further details because functional programming is not the main focus of this thesis.
4.9.1 Functions in Scala

Scala is conceptually an object-functional programming language and thus combines aspect of both programming paradigms. For instance, in Scala functions are objects too. They can be passed around like any other value, assigned to variables and can be stored in data structures. Scala does not distinct a function from a method and thus the syntax is identically.

In this context, the term *method* refers to the technical implementation in Scala, whereas the term *function* is used as defined in definition 4.10.4. The practical implication is that a function is essentially method with the property of determining the output entirely by the function itself based on the parameter but without modifying the actual value of the parameter. In contrast, a *normal* method would modify the parameter value and may determine the return value based on other, even external, factors.

A basic example\(^{48}\) function is shown in listing 4.50. The object keyword creates a new singleton instance that contains two basic methods (*abs*, *formatAbs*) and a main method. The *abs* method represents a pure function that means there are no side-effects. The *abs* method takes an integer and returns its absolute value. The *formatAbs* method is also a pure function which takes an integer as parameter formats it to a string and returns the formatted string to the calling method. The *formatAbs* method is private which means it can only be used within the scope of the instance of MyModule. The main method finally is an *outer shell* that handles the effect of printing to the console(2 in [7]).

```scala
object MyModule {
  def abs(n: Int): Int = if (n < 0) -n else n
  private def formatAbs(x: Int) = {
    val msg = "The absolute value of %d is %d"
    msg.format(x, abs(x))
  }
  def main(args: Array[String]): Unit = println(formatAbs(-42))
}
```

**Listing 4.50:** Simple function in Scala

Scala supports polymorphic functions \(^{63\hphantom{6}}\)(2.6 in [7]) using the same syntax and rules as type parameters in generic methods (see 4.4.2).

\(^{48}\hphantom{48}(2.2\hphantom{2} in [7])\)
4.9.2 Higher ordered functions

When writing purely functional programs in Scala, common practice is to combine functions by simply passing a function as parameter to other functions. A function that takes another function as argument or returns a function is also called a higher order function (2.5 in [7]).

Sometimes, a function is better stored inside another function, which is called an inner or local function [63]. Higher ordered functions and nested functions fit well together and allow a fine grained decomposition of functions. The example\(^{49}\) in listing 4.51 shows the higher ordered function `formatResult` that takes a string, an integer and a function as parameter. The function given as parameter must be of type `f : Int ⇒ Int` which means the function takes an integer and returns another integer.

Inside the function body, an immutable string is created using a format pattern. Here, `%s`, `%d` and `%d` are place holders and will be substituted by arguments passed to the format method. The format method is a standard library method defined on String. The format method is called by passing the name, integer value and the function `f` applied to the integer value.

Since the function `f` is a parameter, the higher order function `formatResult` is pure because the return value (formatted string) is determined only by the function itself and does not modify any of the passed parameters. The main method defines again an outer shell that handles the effect of printing to the console\(^{2.5}\) in [7]).

```scala
object MyModule {
  def formatResult(name: String, n: Int, f: Int ⇒ Int) = {
    val msg = "The %s of %d is %d."
    msg.format(name, n, f(n))
  }
  def main(args: Array[String]): Unit = {
    println(formatResult("absolute value", -42, abs))
  }
}
```

Listing 4.51: Higher ordered function in Scala

\(^{49}\) (2.5 in [7])
4.9.3 Anonymous functions

Scala provides a lightweight syntax for defining nameless functions which are called
an anonymous functions or lambda expression. An anonymous function can be nested
inside any other function or placed inside a function call. Because of Scala’s expres-
sive syntax, there are many different ways to define an anonymous function. The
example\(^{50}\) in listing 4.52 shows five different ways to write an anonymous function
that simply increments an integer value by one. The anonymous function is passed
as a parameter to the `formatResult` function previously defined in listing 4.51.

The first version denotes a complete function signature, the parameter and its type
as well as the function definition. The function declaration is denoted by using the
’⇒’ arrow. The second versions omits the type annotation as type inference\(^{51}\) in
Scala can determine the actual type (Int) of x based on the body and the final return
type of the function.

```scala
object MyModule {
  def main(args: Array[String]): Unit = {
    println(formatResult("increment1", 7, (x: Int) => x + 1)) // (1)
    println(formatResult("increment2", 7, (x) => x + 1))    // (2)
    println(formatResult("increment3", 7, x => x + 1))     // (3)
    println(formatResult("increment4", 7, _ => 1))         // (4)
    println(formatResult("increment5", 7, x => { val r = x + 1; r })) // (5)
  }
}
```

**Listing 4.52: Anonymous function in Scala**

The third version omits the braces which valid in Scala as long as statements can be
interpreted. The statement can be interpreted because the function is fairly simple
and does not have long parameter lists or complex statements in the function body.
The fourth version is the shortest possible notation in Scala and translates to `Any`
(denoted by underscore) gets incremented by one. Even though this notion is very
compact, it has the shortcoming of minimizing type-safety as `Any` is (similar to
Java.Lang.Object) the super type of all values and references in Scala. This function
may fail if a type is passed that cannot be incremented. The last version is rather
verbose compared to other but it follows the definition of a pure nested function in
the sense that it does not alter the parameter and has no side effect\(^{(2.5.1 \text{ in } [7])}\).

\(^{50}\)(2.5.1 in [7])
\(^{51}\)(see section 4.4.1)
4.9.4 Higher kinded types

A specialized function that takes a type as parameter and returns a (different) type is known as higher kinded type. Assuming a container interface that is implemented by different containers for several types of data. The interface should define access to the values in these containers without specifying the values actual type.

Listing 4.53 shows the container interface implemented as higher kinded type. A second type parameter is required to determine the generic structure of the type. The Container type signature \( [M[\_\_]] \) refers \( M \) to the type of the container and the underscore refers to \( \text{Any} \) type that can be placed inside a container of type \( M \). The put and get methods specifying parameter and return type \( \text{Any} \) (denoted with underscore) further to type \( A \) for a container of type \( M[A] \).

The practical implication is the simplicity of using this interface. It can be implemented multiple times for lists, maps, vectors and so on. But it can also be done ad-hoc by creating a new instance implemented with an anonymous nested class as the second part of the example\(^{52}\) shows in listing 4.53.

```scala
trait Container[M[_]] {
  def put[A](x: A): M[A]
  def get[A](m: M[A]): A
}

using REPL
scala> val container = new Container[List] {
    def put[A](x: A) = List(x);
    def get[A](m: List[A]) = m.head
  }
  \"\ using the container
container.put(123)
res: List[Int] = List(123)
```

**Listing 4.53**: Higher kinded types in Scala

\(^{52}\)http://twitter.github.io/scala_school/advanced-types.html
4.10 Functional composition

Monads and monoids are important concepts for combining functions in functional programming. More recently (2012), the inclusion of future and promises [97] in Scala has led to an increased application of monadic design patterns in order to decompose tasks for concurrent execution. This is particularly useful in order to simplify the development of scalable applications [98].

This section is divided into three parts. First, it provides basic definitions in order to establish the theoretical foundation required for the covered topics. Second, monoids and monads are presented together with some examples. Third, the concept of monadic composition is presented and applied to a commonly known problem.

4.10.1 Definitions

A) Preliminaries

Definition 4.10.1 (Set). A set is a mathematical entity that is distinct from, but completely determined by its elements (if any). For every entity $x$ and set $S$, the statement $x \in S$, is either true or false.

A finite set can be denoted by listing its elements with braces, for instance the set \{1, 2, 3\} is determined by the fact that its only elements are the numbers 1, 2 and 3. The empty set is a set with no elements and denote as $\emptyset$.(1.1.1 in [99])

Definition 4.10.2 (Cartesian product). If $S$ and $T$ are sets, the cartesian product of $S$ and $T$ is the set $S \times T$ of ordered pairs $(s,t)$ with $s \in S$ and $t \in T$. Observe that $T \times S = \{(s,t)\mid t \in T \text{ and } s \in S\}$ and so is not the same set as $S \times T$ unless $S = T$. (1.1.4 in [99])

Definition 4.10.3 (Relation). A relation $\alpha$ from a set $S$ to a set $T$ is a subset of $S \times T$. Any such subset is a relation from $S$ to $T$. The usual order relations such as "<" and "≤" on rational numbers are examples of relations. Observe that if $S \neq T$, then a relation from $S$ to $T$ is not the same as a relation from $T$ to $S$ because $S \in T \neq T \in S$. (1.1.5 in [99])
B) Function definitions

**Definition 4.10.4** (Function). A function $f$ is a mathematical entity with the following properties:

- **F-1** $f$ has a **domain** and a **codomain**, each of which must be a set.
- **F-2** For every element $x$ of the domain, $f$ has a **value** at $x$, which is an element of the codomain and is denoted $f(x)$.
- **F-3** The domain, the codomain, and the value $f(x)$ for each $x$ in the domain are all determined completely by the function.
- **F-4** Conversely, the data consisting of the domain, the codomain, and the value $f(x)$ for each element $x$ of the domain completely determine the function $f$.

The domain and codomain are often called the **source** and **target** of $f$, respectively. The notation $f : S \to T$ is a succinct way of saying that the function $f$ has domain $S$ and codomain $T$. The significance of **F-3** is that a function is not merely a rule, but a rule together with its domain and codomain.

We will use the **barred arrow notation** to provide an anonymous notation for functions. The barred arrow goes from input datum to output datum and the straight arrow goes from domain to co-domain. For example, the function from $\mathbb{R}$ to $\mathbb{R}$ that squares its input can be denoted as:

$$x \mapsto x^2 : \mathbb{R} \to \mathbb{R}$$

The barred arrow notation is used mainly for functions denoted by a formula. (1.1.6 in [99])

**Definition 4.10.5** (Referential transparency). An expression $e$ is referentially transparent if for all programs $p$, all occurrences of $e$ in $p$ can be replaced by the result of evaluating $e$, without affecting the observable behaviour of $p$ (1.2 in [7]).

**Definition 4.10.6** (Function purity). A function $f$ is pure if the expression $f(x)$ is referentially transparent for all referentially transparent $x$ (1.2 in [7]).

---

$^{53}$\text{R} denotes the field of rational numbers
C) Monad and monoid definitions

The next definitions are required to establish the foundation for Monads and monoids before illustrating the actual usage in Scala.

**Definition 4.10.7** (Semigroup). A semi-group is a set $S$ together with an associative binary operation $S \times S \rightarrow S$. The set $S$ is called the underlying set of the semi-group. (2.3.4 in [100])

**Definition 4.10.8** (Identity element). An identity element $e$ for a semi-group $S$ is an element of $S$ that satisfies the equation:

$$se = es = s \forall s \in S$$

There can be at most one identity element in a semi-group (2.3.7 in [100]).

**Definition 4.10.9** (Monoid). A monoid is a semi-group with an identity element. It is commutative if its binary operation is commutative. It follows from the definition that a monoid is not allowed to be empty: it must contain an identity element (2.3.8 in [100]).

**Definition 4.10.10** (Homomorphisms). If $S$ and $T$ are semi-groups, a function $h : S \Rightarrow T$ is a homomorphism if for all $s \in S, h(ss) = h(s)h(s)$ (2.5.1 in [100]).

A homomorphism of monoids is a semi-group homomorphism between monoids that preserves the identity elements: if $e$ is the identity element of the domain, $h(e)$ must be the identity element of the codomain (2.5.1 in [100]).

**Definition 4.10.11** (Monad). A monad is defined as a triple $T = (T, \eta, \mu)$ on a category $A$ consists of a functor $T : A \Rightarrow A$, together with two natural transformations: 1) $\eta : id \Rightarrow T$

2) $\mu : T^2 \Rightarrow T$

The transformation $\eta$ is the unit and $\mu$ is the multiplication (14.3 in [100]).

---

54 A binary operation on a non-empty set $S$ is a function which sends elements of the Cartesian product $S \times S$ to $S$: $f : S \times S \Rightarrow S$.

55 A binary operation $*$ on a set $S$ is called commutative if: $x * y = y * x$ for all $x, y \in S$.
**Definition 4.10.12 (Category).** A category is a graph $\delta$ together with two functions: $c : C^2 \to C_1$ and $u : C_0 \to C_1$ with properties $C_1$ through $C_4$ below. The elements of $C_0$ are called objects and those of $C_1$ are called arrows. The function $c$ is called composition, and if $(g, f)$ is a composable pair, $c(g, f)$ is written $g \circ f$ and is called the composite of $g$ and $f$. If $A$ is an object of $\delta$, $u(A)$ is denoted $id_A$, which is called the identity of the object $A$.

- **C-1** The source of $g \circ f$ is the source of $f$ and the target of $g$
- **C-2** $(h \circ g) \circ f = h \circ (g \circ f)$
- **C-3** The source and target of $id_A$ are both $A$.
- **C-4** If $f : A \Rightarrow B$ in $C$, then $f \circ id_A = id_B \circ f = f$

The significance of the fact that the composite $c$ is defined on $G_2$ is that $g \circ f$ is defined if and only if the source of $g$ is the target of $f$. This means that composition is a function whose domain is an equationally defined subset of $G_1 \times G_1$; the equation requires that the source of $g$ equal the target of $f$. It follows from this and $C1$ that in $C_2$, one side of the equation is defined if and only if the other side is defined (2.1.3 in \cite{100}).

**Definition 4.10.13 (Functor).** A functor $F$ from a category $B$ to a category $D$ is a graph homomorphism which preserves identities and composition. It plays the same role as monoid homomorphism for monoids and monotone maps for posets: it preserves the structure that a category has. The definition is:

A functor $F : B \Rightarrow D$ is a pair of functions $F_0 : B_0 \Rightarrow D_0$ and $F_1 : B_1 \Rightarrow D_1$ for which

- **F-1** If $f : A \Rightarrow B$ in $C$, then $F_1(f) : F_0(A) \Rightarrow F_0(B)$ in $D$.
- **F-2** For any object $A$ of $C$, $F_1(id_A) = id_{F_0(A)}$.
- **F-3** If $(g \circ f)$ is defined in $B$, then $F_1(g) \circ F_1(f)$ is defined in $D$ and $F_1(g \circ f) = F_1(g) \circ F_1(f)$

By $F_1$, a functor is in particular a homomorphism of graphs. By $F-2$ preserves the identify and $F-3$ defines the preservation of composition (3 - 3.1 in \cite{100}).

---

$^57$ $C_2$ is the set of paths of length $2$
4.10.2 Monoids in Scala

A monoid implemented in Scala consist of:

O-1 Some type A

O-2 A binary associative operation that takes two values of type A and combines them into one result value of type A.

O-3 A value of type A that is an identity for that operation.

The laws of associativity and identity are collectively called the monoid laws. Defining a monoid in Scala requires the specification of these monoid laws which can be done in an abstract class as it is shown in the example in listing 4.54. Any specific implementation of this monoid trait is by definition, composable because of the binary associative operation provided (10.5 in [7]). This also allows certain performance optimization such as associate transformation into pair data structures for concurrent execution (10.3 in [7]). Furthermore, generic functions can be writing that knowing nothing about a specific type other than that it’s a monoid.

```scala
abstract class SemiGroup[A] { def add(x: A, y: A): A }
abstract class Monoid[A] extends SemiGroup[A] { def unit: A }
```

Listing 4.54: Monoid interface in Scala

Dealing directly with monoids can be difficult in practice. As Scala supports implicit conversions (see 4.5, any kind of type can be converted implicitly into a monoid to gain the implied benefits. The major benefit is that the public interface of an affected method remains unchanged while adding parallel execution through implicit monad conversion(10.5.3 in [7])

The example in listing 4.55 illustrates a String and an Int monoid. At the first sight, implementing a Monoid is just like implementing any other generic interface in Scala. Only the type needs to be overridden and the operations needs to be type specific defined.

---

58(10.1 in [7])  
59http://www.scala-lang.org/node/114  
60http://www.scala-lang.org/node/114
Using implicits adds three important details to the implementation shown in listing 4.55. First, implicit objects are used to perform an implicit conversion from a type T into a Monoid[T]. For instance, from String to Monoid[String]. However, the conversion can only be performed for types if a corresponding conversion method or implicit object is present. The example can convert Int and Double but not, for instance, Boolean (10.5.3 in [7]).

Second, the sum function requires its second parameter to be a monoid of the same type as the passed list. As the parameter is implicit, the compiler will attempt to create an instance for the target type by using the corresponding implicit object.

Third, the actual sum is a recursive call of the sum function itself which expands to a nested addition of all elements of the list. By using the add function of the implicitly created monoids, the correct add function will be used for each type.

```
object ImplicitTest {
  implicit object StringMonoid extends Monoid[String] { // (1)
    def add(x: String, y: String): String = x concat y
    def unit: String = ""
  }

  implicit object IntMonoid extends Monoid[Int] {
    def add(x: Int, y: Int): Int = x + y
    def unit: Int = 0
  }

  def sum[A](xs: List[A])(implicit m: Monoid[A]): A = {
    if (xs.isEmpty) m.unit
    else m.add(xs.head, sum(xs.tail)) // (3)
  }

  def main(args: Array[String]) {
    println(sum(List(1, 2, 3)))
    println(sum(List("a", "b", "c")))
  }
}
```

**Listing 4.55:** Monoid implementation in Scala

Using monoids illustrates three important benefits. First, generic functions such as sum are simpler to implement as the type specific add function will be dispatched dynamically to the type specific one. Applying this mechanism to categories of types, allows ordinary functions to be written exactly once using the monoid pattern(10.5 in [7]).
Second, as example already indicates, implicit conversions allow to shield the complexity of using monoids from public interfaces. A thin wrapper that calls private functions using implicit conversions hides the internal representation entirely.

Third, monoids are composable using binary associative operation on arbitrary objects representable as monoid. For instance, the example\(^{61}\) in listing 4.56 illustrates the composition of arbitrary case classes (see 4.3.6) that are implicitly converted to monoids by using the \texttt{shapeless}\(^{62}\) library. Using shapeless minimizes the overhead required to transform case classes into monoids by providing a convenient conversion methods that requires just two steps.

First, the \texttt{HListIso} needs to be published. A \texttt{HListIso} is a ”Heterogeneous List Isomorphism”. Essentially all generic operations in shapeless are implemented by using a heterogeneous list, also known as \texttt{HList}, that is a poly-typed list allowing operations on elements of different types. Adding an isomorphism to a \texttt{HList} by declaring the injector (apply) and extractor (unapply) methods (see 4.2.2 for details) makes an additional type available to a \texttt{HList}. Second, using the \texttt{implicitly} keyword\(^{63}\) converts a case class into a monoid which then can be added using the \texttt{| + |} operator similar to addition on primitive data types.

```scala
object MonoidExamples extends App {
  // Arbitrary case class
  case class Foo(i : Int, s : String)
  // 1) Publish HListIso
  implicit def fooIso = Iso.hlist(Foo.apply _, Foo.unapply _)
  // 2) And now they’re monoids
  implicitly [Monoid[Foo]]
  // Actual usage:
  val x = Foo(13, "foo")
  val y = Foo(23, "bar")
  val f = x |+| y // Adding two monoids together
  assert(f == Foo(36, "foobar"))
}
```

Listing 4.56: Monoid case classes in Scala

Monoids are already useful but somewhat limited by allowing only binary associative operations. Sometimes, more expressiveness is needed which requires monads.

\(^{61}\)https://github.com/milessabin/shapeless/tree/master/examples
\(^{62}\)https://github.com/milessabin/shapeless
\(^{63}\)see 4.5 for details on implicit conversions
4.10.3 Monads in Scala

A monad is a functor defined on categories which allows a much broader application. The important difference is that monoids can only be applied to binary associative operations but monads are using functors instead. According to definition 4.10.13 a functor is a homomorphism\textsuperscript{64} which preserves identity and composition(11.1 in [7]). The signature of a functor is defined as $F : B \Rightarrow D$ which means, a functor maps elements from $B$ to elements of $D$ as long as $B$ and are of the same category. Thus, the name of the function of a functor is map in Scala. A trait defining a functor in Scala is shown in listing 4.57.

``` scala
trait Functor[F[_]] {
  def map[A,B](fa : F[A])(f : A => B) : F[B]
}
```

Listing 4.57: Functor interface in Scala

Note, the type parameter of trait Functor is $[F[_]]$ which means a container of type F must contain any type. The map function itself is a type signature with two type parameters (A and B) and two parameter lists. The first parameter list specifies the actual type of the $[F[_]]$ signature further by defining it must be of type $[F[A]]$. The second parameter is a function of that maps type A to type B. The return type requites to be $[F[B]]$. The functor trait in listing 4.57 fully complies with definition 4.10.13 and thus allows specifying a monad in Scala(11.1 in [7]).

Similar to monoids, monads have laws:

M-1 A monad are associative composable.

M-2 A monad has associative identity\textsuperscript{65}.

Implementing a monad in Scala requires three steps. First, an identity needs to be defined, second, a generic map needs to be defined and third, a composition function needs to be defined.

\textsuperscript{64}The word homomorphism comes from Greek: "homo" means "same" and "morphē" means "shape". Thus it preserves the same shape.

\textsuperscript{65}In functional programming, this is sometimes called left and right identity.
The last one is only required because Scala does not have a predefined compose function. Listing 4.58 shows a monad trait that extends the `Functor` trait from the previous listing 4.57. A monad in Scala provides a standard interface for composing and sequencing operations on value(s) contained in a monad. Thus, the monad interface defines an identity and three functions: `map`, `flatMap` and compose. In order to preserve associativity, the `Unit` function defines identity as the argument itself. The `flatMap` applies a monadic function\(^{66}\) to the contained value(s) in a monad. The `map` function applies a regular function to the contained value(s) by passing a monad and the function to `flatMap` which composes them into one result. Likewise, the `compose` function takes two monads and a function and composes them into a third monad using `flatMap` [101](11.2.1 in [7]).

```scala
trait Monad[M[_]] extends Functor[M] {
  def unit[A](a: => A): M[A]
  def flatMap[A,B](ma: M[A])(f: A => M[B]): M[B]
  def map[A,B](ma: M[A])(f: A => B): M[B] = flatMap(ma) (a => unit(f(a)))
  def compose[A, B,C](f: A => M[B], g: B => M[C]): A => M[C] =
    a => flatMap(f(a))(g)
}
```

**Listing 4.58:** Monad interface in Scala

### 4.10.4 Monadic composition

Applying monads in practice allows compact solutions for rather complex problems. Considering the following scenario, an online shop may want to speed up the loading time of the shopping page. The idea is to decouple data fetching from page rendering to allow asynchronous processing for each task. One task loads the required data from the database and another one renders the result in HTML. A common pitfall is the issue of delayed page loading due to slow database queries caused by high high system load. To cope with this issue, data querying and page rendering should be split in independent tasks in order to improve overall performance. Next, dividing HTML rendering in smaller sub-tasks allow concurrent execution so that the rendered results will be combined together before updating the shopping site.

\(^{66}\)that is a functor
Using monads simplifies the construction of such a work-flow. Beginning with the data transaction, the monad from listing 4.59 is used to define transactable data types. Only three important details needs to be clarified. First, instead of a real JPA\textsuperscript{67} EntityManager an empty place holder is just because the example illustrates the concept of monadic composition. Second, the monad Transactional defines the identity as Unit instead of a of type A. This is because the actual type is wrapped into an EntityManager and regardless of type A, the identity of EntityManager is always Unit in Scala. Third, to create specific transactional monads, a companion object is used.

\begin{Verbatim}
class EntityManager { def find : Nothing // (1) Placeholder for JPA EntityManager
trait Transactional[A] {
  def unit => Unit  // (2) Unit identity
  def atomic : EntityManager => A
  def map[B] (f : A => B) : Transactional[B] = transactional[B](f compose atomic)
  def flatMap[B] (f : A => Transactional[B]) : Transactional[B] =
  transactional {r => f(atomic(r)).atomic(r)}
object Transactional { // (3) companion object
  def transactional[A] (em : EntityManager => A) {
    = new Transactional {def atomic = em}
  }
}
\end{Verbatim}

\textbf{Listing 4.59:} Transactional monad in Scala

In order to apply the transactional monad for the illustrated use-case is shown in listing 4.60. The first function \texttt{findByPK} is used to fetch a product by its primary key from the database. To do so, an anonymous nested function is passed to the companion object of the transactional monad which then creates a new transactional monad, performing the query and returns the result. The second function \texttt{renderShoppingBasket} wraps the Transactional signature on type String to ensure only a completely rendered result will be returned. The third function \texttt{renderProduct} passes, similar to the first one, passes an anonymous nested function that takes anything (in this case any product) and passes it to the \texttt{renderShoppingBasket} basket which creates a transactional monad and applies the rendering function to the product.

Finally, bringing everything together, the work-flow of the application is shown in listing 4.61. The entire code is asynchronous and non-blocking. Thus, any delay from one task does not affect any of the other tasks.

\textsuperscript{67}Java Persistence API
Chapter 4. Scala

Listing 4.60: Monadic combination Scala

For listing 4.61, five important observations need to be clarified. First, the example uses Scala’s for-comprehension, as explained in the next section about concurrency. For the example, it is enough to know that for-comprehension executes monadic operations concurrently. Second, the similarity of `findProdByPK (id)` and `findBaskByPK (id)` is no coincident, both actually refer to a generic function `findByPK` but because of the explicit specialization, they can be executed independently from each other. Third, the `renderProduct` function gets executed in the moment the actual product is available. Next, `renderShoppingBasket` is rendered independently from the product page. Fifth, the rendered results of the product and the basket page are finally combined together in the last step. This is possible because of the usage of the String transactor monad that allows the combination operation.

Listing 4.61: Monadic combination Scala

This example provides already a first impression how functional composition and concurrency work together in Scala. The next section provides more about concurrency in Scala.
Chapter 4. Scala

4.11 Scala concurrency

Concurrency in Scala, unlike many other mainstream programming languages such as Java or C#, Scala does not require working directly with threads or locks. Consequently, fundamentals in concurrency programming are not covered in this section for three reasons. First of all, the reader is assumed to be familiar with fundamental concepts of concurrency such as threads, locks and blocking. Thus, only a brief definition of commonly used terms is given in this section. Second, concurrency programming in Java is already well covered in existing literature [102][103], textbooks [11][104] and extensively criticised [105][106][107]. The major problems of Java’s standard concurrency are the usage of shared-memory threads and mutable states that are complicated to program, requires locks and suffer from high initialization and context-switching overhead as well as high memory consumption [108]. Third, the primary scope of this thesis is related to concurrency in Scala and not Java.

Instead, the three main abstractions for concurrency in Scala are covered in this section. These concepts are parallel collections, futures and actors. Even though concurrency in Java is not part of this thesis, a brief introduction to the Disruptor framework written in Java is provided. The choice for the disruptor framework is entirely based on the fact that the disruptor is known\textsuperscript{68} to be the fastest event based concurrency implementation available for Java. Therefore, it is the first choice in order to draw a comparison between Scala’s built-in concurrency capabilities versus one state of the art framework.

The first abstraction is parallel collections that provide for each linear collection a parallel collection. Since collections are a well-understood and a frequently-used programming abstraction, the idea is to use implicit parallelism in order to bring reliable parallel execution into mainstream development.

The second abstraction is known as composable futures. A Future is a data structure used to retrieve the result of a concurrent operation which may get available later. A method call on a future gets suspended until the passed computation is either available or timed out. In practice, a future wraps any method and shields the developer from the actual concurrent execution. As a future is structurally a monad, futures can be nested or combined while being executed concurrently.

\textsuperscript{68}As presented in Java Magazine April issue 2012
The third abstraction are actors which can be used either for concurrent execution on one system or for parallel execution in a distributed environment. Scala follows the Erlang actor model that avoids sharing state among threads entirely by only passing immutable messages between actors. All three concepts are already part of the standard Scala distribution and can be used without any additional configuration. In order to accomplish the high level of abstraction, concurrency in Scala is based on two concepts: polymorphic delimited continuations and adaptive work-stealing.

The structure of this section is as follows: First some frequently used terms are briefly defined to clarify the exact meaning. Second, the underlying concepts of delimited continuations and adaptive work stealing are covered. This is followed by the three main concurrency abstractions in Scala: parallel collections, composable futures and actors. Finally, the Disruptor framework is briefly introduced and illustrated on an example how to use the framework in practice.

\[69\] Erlang is a dynamically typed functional programming language designed for distributed, fault-tolerant, soft-real-time control systems in telecommunication [109][110].

\[70\] That is analogous to the JDK libraries
4.11.1 Concurrency terminology

Concurrency vs. Parallelism

"Concurrency and parallelism are related concepts, but there are small differences. Concurrency means that two or more tasks are making progress even though they might not be executing simultaneously. [...] Parallelism on the other hand arise when the execution can be truly simultaneous.” (2.1.1 in [111])

Asynchronous vs. Synchronous

"A method call is considered synchronous if the caller cannot make progress until the method returns a value or throws an exception. On the other hand, an asynchronous call allows the caller to progress after a finite number of steps, and the completion of the method may be signalled via some additional mechanism (it might be a registered callback, a Future, or a message).” (2.1.2 in [111])

Non-blocking vs. Blocking

"We talk about blocking if the delay of one thread can indefinitely delay some of the other threads. A good example is a resource which can be used exclusively by one thread using mutual exclusion. If a thread holds on to the resource indefinitely (for example accidentally running an infinite loop) other threads waiting on the resource cannot progress. In contrast non-blocking means that no thread is able to indefinitely delay others.” (2.1.3 in [111])

Task parallelism

Tasks task parallelism is achieved when each processor executes a different thread on the same or different data (5 in [112]).

Data parallelism and SIMD

In data parallelism each processor performs the same task on different pieces of distributed data. SIMD refers to a single instruction processing multiple data in parallel which is an efficient and fast way of data parallelism (4 in [112]).
4.11.2 Delimited continuations

Continuations in classical programming are control structures that are used to modify the program-flow during execution. Complete continuations use a specific keyword to mark an alteration of program flow and from there on, embody the entire rest of the computation after the keyword. Delimited continuations are partial and composable continuations that behave more like regular functions. Unlike complete ones, delimited continuations will actually return control to the caller after invocation and they may also return values. This means that delimited continuations can be called multiple times in succession, and the program can proceed at the call site afterwards [113].

In practical applications, delimited continuations are a mechanism to receive, suspend and resume any given task without syntactic overhead and without being tied to JVM threads. One important observation is that delimited continuations are able to express any definable monad and thus allow composition of tasks that can be suspend and resumed at any time [113][114][115]. Scala provides support for delimited continuations by two new keywords enabling direct-style shift and reset with full type polymorphism and statically checked through the compiler [113].

Delimited continuations were previously only available in niche programming languages such as OCaml and Scheme. Through the direct language support in Scala, delimited continuations are used to make previously less popular concurrency concepts accessible to a broader audience. In particular, actors and futures are implemented very efficiently using delimited continuations which leads to a compact syntax at an attractive performance in concurrent execution time [113].
4.11.3 Adaptive work-stealing

Scala’s collections provide a rich set of pre-defined methods for manipulating stored elements. For Scala’s parallel collections, those methods are implemented on top of a parallel `foreach` method that assigns a subset of elements to different processors. Each time a user invokes the `foreach` method by calling methods of the parallel collection, a thread would be created and assigned subset of elements to work on.

As thread creation and initialization is an expensive operation, it makes sense to use a pool of worker threads in sleeping state and avoid thread creation. Scala parallel collections uses Java’s Fork/Join Framework to manage a pool of worker threads, spawn new tasks (fork) and later wait for them to finish (join).

The fork/join pool abstraction can be implemented in several different ways, including work stealing as the Java version does. Work stealing means each processor maintains a task queue and dequeues one task after the other for processing. If the queue becomes empty, the processor tries to steal a task from another processors queue. For work stealing to be effective work must be partitioned into tasks of small enough granularities, which implies a certain overheads if there are too many tasks (3 in [8]).

There are two important observations. First, stealing tasks is generally more expensive than dequeuing them from the task’s own queue. Second, the fork/join framework allows only the oldest tasks on the queue to be stolen. The implications are two fold: Less work stealing is desirable because of the implied cost thus stealing bigger tasks reduces the overhead. Second, what task gets stolen depends on the order tasks were pushed to the queue which needs to be considered for delegating tasks to the fork-join pool. In order to address these two issues, Scala uses exponential task splitting [116] for distributing tasks to the fork-join pool (3 in [8]).

The general concept of exponential task splitting is that for each parallel operation, the collection is split into two parts. For one of these parts, a task is created and forked which means the task gets pushed on the processors task queue. The second part gets split in the same way until a threshold is reached which means that subset of elements in the collection is operated on sequentially. This iterative splitting divides a task into exponential smaller ones. Once a processor has finished a task, it pops a task of its queue, in case the queue has further elements.
Figure 4.5 illustrates exponential task splitting compared to normal, fine grain, task splitting as shown in figure 4.5a. It is important to note that a task is getting smaller with each splitting iteration. All sub-tasks are pushed to the queue after splitting has finished. This means, the last (smallest) task pushed will be the first task popped from the queue. At any time a processor tries to pop a task, it will be assigned an amount of work equal to the total work done since it started with the leaf (3 in [8]). If a processors queue is empty, the task will steal from the opposite side of the queue which means the biggest task gets stolen first. When a processor steals a task, it divides the subset of the collection assigned to that task until it reaches threshold size of the subset. Stolen tasks are divided into exponentially smaller tasks until a threshold is reached and then handled sequentially starting from the smallest one, as shown in figure 4.5b. On the other hand, tasks that came from the processors own queue are handled sequentially straight away (3 in [8]).

One important aspect is that the maximum number of created tasks depends on the chosen threshold. In order to scale with the number of processors, parallel collections determine the threshold based on the number of elements in a collection. More precisely, the threshold is defined as the number of elements divided by eight times the number of processors. This means the number of created tasks will be one order of magnitude greater than the number of processors if no work stealing occurs which scales very well in practice (5 in [8]).
4.11.4 Parallel collections

Scala’s parallel follow implement unified collections which means for each type of collection, a parallel version is also available \[][8][9]. Unified collections in Scala\(^{71}\) implements a particular hierarchy of abstraction as illustrated in figure 4.6. The \textit{Traversable} trait is a common ancestor of all collection classes. \textit{Traversable} defines an abstract method \textit{foreach}, which traverses all of the elements of the collection and applies a specified higher-order function to each element.

Trait \textit{Iterable} is a descendant of \textit{Traversable} and declares an abstract method \textit{iterator} which returns an iterator used to traverse elements. The difference to \textit{foreach} is, an iterator requires the next element to be requested explicitly and not all elements have to be traversed as \textit{foreach} would do \[8\] (2 in \[9\]).

![Abstraction hierarchy in Scala’s collection](image)

\textbf{Figure 4.6: Abstraction hierarchy in Scala’s collection \[9\].}

Three other traits inherit from \textit{Iterable}: \textit{Set}, \textit{Seq} and \textit{Map}. \textit{Seq} is the shared supertype of collections where each element such as \textit{Array} or \textit{List}. Trait \textit{Set} denotes \textit{Iterable} collections which does not contain duplicate elements. It defines abstract methods for adding and removing elements from the set, and checking if an element is already in a set. \textit{Maps} are \textit{Iterable} collections of pairs of keys and values. They define abstract methods for adding and removing entries to the map, and the get method used to lookup values associated with keys.

\(^{71}\)See 4.3.1 for details about unified collections
Some operations on collections return another collection thus trait *Builder* is used to build the required type of collections. Trait *Builder* generalizes the concept by taking the element type of the collection and the target collection type as parameter and returns the requested collection. Each collection provides a builder implementation for creating an instance of the collection or for converting another collection into the target type.

Scala’s standard collections are inherently sequential which means one element after the other needs to be accessed in linear order thus abstract *foreach* and *iterator* are both strictly linear operations. In order to enable parallel collections, operations needs to be split in smaller parts, executed concurrently and collected to a final result. Implementing the required split and collect methods for each collection individually would imply duplicated code which is difficult to maintain. Instead, Scala provides a generic parallel framework applicable for all collection. Following the split-combine concept, two new abstractions are introduced: *splitter* and *combiner* (4 in [8]).

A splitter is an iterator with standard methods such as *next* and *hasNext* used to iterate elements of a collection. It has an additional method *split* which splits a splitter into subSplitters which each traverse over disjoint subsets of elements.

A combiner has a method *combine* that takes another combiner and produces a combiner that contains the union of their elements. It is important to note that a combiner is created always in correspondence with a splitter to ensure the exact reversal of the splitting order. A combiner takes two type parameters which denote the type of the contained elements and the type of the resulting collection.

Using splitter and combinators allows the implementation of *ParIterable* which is the base trait for all parallel collections. Trait *ParIterable* has the same operations as the *Iterable* trait for sequential collections with the only difference of having an abstract method *parallelIterator* and *newCombiner*. Method *parallelIterator* returns a splitter for the collection and method *newCombiner* returns the combiner for the collection. All operations for parallel collections are implemented in terms of splitters and combinators which means, all collection operations are concurrent by default [8].

The direct consequence of this approach is an out-of-order execution of sub-tasks but an ”in order” combination because a combiner reverses the splitting order.
However, the "out-of-order" semantics of parallel collections lead to the following two implications:

1) Side-effecting operations can lead to non-determinism
2) Non-associative operations lead to non-determinism

Side-effects are generally problematic in concurrent execution because they can lead to race conditions. Race conditions occur when two (or more) threads attempt to re-assign the same variable. A race condition can only be prevented by careful threat looking and synchronized field access. Applying the concept of function purity (section 4.9) removes the problem of such side-effects entirely.

Non-associative operations are a serious problem because of the out-of-order execution of sub-tasks. As the associative law requires, the order of operations does not affect the final results. A non-associative operation thus cannot be ensured to return the same result because the operational order is not guaranteed.

Even though Non-associative operations are a known limitation, all operations are still commutative because of the exact reversal of the splitting order. More precisely, if a task is split into three sub-tasks (A, B, C) it will be reassembled once again in exactly the same order A, B, C. As the order does not change, the commutative law remains true on any parallel operation. Consequently, as the commutative property is given, any associative operation can also be applied to the distribute law. This means the results from several parallel operations can be distributed again.

The parallel collection architecture not only mirrors the already existing sequential collections but also fully preserves the corresponding API. The implication of this approach is that any sequential collection can be "swapped out" for its parallel counterpart by changing the collection type from sequential to parallel. This means, existing applications can use parallel collections with minimal effort and improve their performance on multi-core architectures. Using parallel collections in practice can be done by either calling a different constructor or by converting any existing collection into a parallel one, as shown in listing 4.62.

\footnote{http://docs.scala-lang.org/overviews/parallel-collections/overview.html}
\footnote{For details on side effects, see section 4.9}
\footnote{Associative law \((a + b) + c = a + (b + c)\)}
\footnote{Commutative law \(a + b = b + a\)}
\footnote{Distribute law \(a(b + c) = ab + ac\)}
// A) using a constructor
import scala.collection.parallel.immutable.ParVector
val pv = new ParVector[Int]
   pv :+ (1,2,3,4,5,6,7,8,9)

// B) Converting sequential to parallel collections
val pv = Vector(1,2,3,4,5,6,7,8,9).par

Listing 4.62: Parallel collection Scala

Experimental results on comparing parallel collections to sequential collections in Scala clearly shows an exponential reduction of run-time with the number of cores. Furthermore, a comparison has been made to Doug Lea’s extra166 parallel array, which is a state of the art extension to Java’s concurrency as defined in JSR166. In both cases, extra166 and Scala’s parallel collections shows an almost identical performance improvement (5 in [8]). The main reason for the overall good performance of Scala’s parallel collection is rooted in the usage of adaptive work stealing and exponential task spiting (see 4.11.3) [8].
4.11.5 Composable futures

Composable Futures in Scala consist of two complementary concepts, a Future and promises. A Future is a place-holder object for a result that does not yet exist but will be available in the future. A promise\textsuperscript{79} on the other hand is conceptually a writeable, single-assignment container, which will be written once a Future is completed. Futures and promises are complementary in the sense that a promise is required to complete a Future and consequently a Future is the result of a promise. The result of a Future may be a value which can be obtained using the \textit{Success} method, or an exception which is accessible by using the \textit{Failure} method\textsuperscript{97}.

The results of several futures are computed independently and can be later collected using for-comprehension. Several Futures are executed asynchronously, non-blocking and concurrently. Using the example\textsuperscript{80} from listing 4.63 illustrates how futures are combined by using for-comprehension. Five observations are shown in the example. First, a promise needs a type. Second, a future can be created just as any other field in Scala which means type inference applies. Third, as \texttt{f2} shows, promise \texttt{p} can be used as well to create a matching future for the promise.

```scala
object testFuture {
  def main() {
    val p = promise[Int] // (1)
    val f = future{42} // (2)
    val f2 = p.future // (3)

    p.completeWith f // (4)
    p.future onSuccess case x => println(x)
    p.future onFailure case e => handle(e)

    f2 onComplete {
      // (5)
      case Success(result) => println(result)
      case Failure(failure) => handle(failure)
    }
  }
}
```

\textbf{Listing 4.63:} Future and promise in Scala

\textsuperscript{79}Promises were first introduced in 1988 by Liskov and Shriira [117].
\textsuperscript{80}http://docs.scala-lang.org/overviews/core/futures.html
Fourth, in case future and promise have been created independently from each other, they need to be matched using the \texttt{completeWith} keyword. Once a future is completed the result is stored in a typed value. Finally, pattern matching is used in order to determine if the typed value is the desired result or an execution failure which means the value contains an exception. This mechanism allows efficient exception handling during run-time \cite{97}.

It is important to note, for Future \( f \), there is no need to create a promise because the Scala compiler creates one implicitly. The keywords \texttt{onComplete}, \texttt{onSuccess} and \texttt{onFailure} denote callbacks that allow reacting to a future completion. As futures are executed concurrently, handling callbacks in sequential order is accomplished by using the \texttt{andThen} keyword.

The \texttt{fallbackTo} keyword chains two futures together into a new Future, and will hold the successful value of the second Future if the first one fails \cite{97}. The example from listing 4.64 illustrates how callbacks are used in Scala.

```scala
object testCallbacks {
  val future1 = loadPage(url)  // (1)
  val future2 = loadPageAgain(url)
  val result = future1 fallbackTo future2  // (2)
  andThen {
    case Failure(exception) => log(exception)  // (3)
    case Success(data) => drawPage(data)  // (4)
  } andThen {
    closeConnection(url)  // (5)
  }
}
```

\textbf{Listing 4.64:} Future and promise in Scala

Five noticeable aspects are shown in the example. First, \texttt{future1} in this case holds the results of a method. Thus, the success of the future depends on the return value of the method. Second, the method tries to load a page but in case the initial attempt fails, the fallback callback is used for executing \texttt{future2} as a second try. Third, using the sequential re-ordering mechanism, the next step is handling any potential exception by logging the error message.

Fourth, using pattern matching again, the \texttt{Success} case is then used to draw the requested page. Finally, using \texttt{andThen} again ensures that the connection will be closed only after logging or page drawing has been finished \cite{97}\cite{118}.  

The major difference to other concurrency approaches is based on the fact that futures are monads. Following the requirements of monads (see 4.10.3 for details) futures defines three essential functions: map, flatmap and filter as well as an identity. This allows futures to be freely combined with other futures, as long as the associative composition is preserved. However, because of the andThen callback, the associative rule required on monads can be relaxed on futures. The functor applied to a monad is simply a method stored inside a future.

Because of the monadic structure, futures are fully composable while being executed asynchronously and concurrently. The main mechanism for future composition is called for-comprehension which is mostly enabled by sophisticated code rewriting by the Scala compiler. Each future in a for-comprehension gets re-written to one of the monadic functions in order to enable monadic composition [97][118].

Example 4.65 shows a simple for-comprehension that executes three futures (x,y,z) concurrently and combines the results together afterwards. As listing 4.66 shows, the body of the yield statement in the for-comprehension is rewritten as the body of a functor passed to map, and the preceding assignments are used to form enclosing flatMaps (5 in [118]). Futures are the foundation for data-flow, covered in the next section.

```scala
object forComprehension {
  val f: Future[Int] = for {
    x <- Future(1+2)
    y <- Future(2+3)
    z <- Future(4+5)
  } yield (x + y + z)
  f onComplete {
    case Success(res) => println(res)
    case Failure(ex) => println(ex)
  }
}
```

```scala
object forRewrite {
  val f = Future(1+2).flatMap {
    x => Future(2+3).flatMap {
      y => Future(4+5).map {
        z => x + y + z
      }
    }
  }
  f onComplete {
    case Success(res) => println(res)
    case Failure(ex) => println(ex)
  }
}
```

Listing 4.65: For-comprehension  Listing 4.66: Compiler rewrite
4.11.6 Actors

Actors are lightweight objects, executed asynchronously and concurrently by using delimited continuations. Actors communicate asynchronously through immutable message passing. The core concept of an actor system is to define one actor per task and one message type per data-structure. A message is sent to an actor that queues all incoming messages and processes all messages in the same order as they arrive. Control structures are defined as patterns of passed messages which means depending on the received message type, an actor takes different actions. Once the received data are processed, the result is sent as another message type to another actor for further processing. Through that mechanism, work flows are composed as message-flows between actors [119].

The actor model was first introduced in 1973 by Hewitt et al. [120] [121] as part of their research in artificial intelligence. The first popular implementation of actors was introduced with the Erlang\(^{81}\) programming language. Erlang is a dynamically typed programming language developed at the Ericsson Computer Science Laboratory\(^{82}\) for building distributed real-time high availability systems such as telephone exchanges [110][109]. Erlang implements concurrent processes by its own runtime system and not by the underlying operating system which gives Erlang a high degree of performance optimization [122].

Scala implements Erlang styles actors by unifying thread and event based paradigms by using, thread-less, event-based actors on non-cooperative virtual machines. Thread-less means, there is no direct coupling between an actor and a thread. Event-based means, an actor reacts only to events such as receiving a message. Non-cooperative refers to virtual machine (JVM) that does not explicitly manage the execution state of a program [123].

Implementing Actors using threads is generally undesired because of the high cost involved in creating threads and context switching. Event based implementations fit actors well as long as inversion of control is avoided since it adds a considerable amount of complexity and thus makes programming more complex and error prone. Scala uses a state of the art unification of thread and event based actor implementation.

\(^{81}\)http://www.erlang.se/

\(^{82}\)http://www.cs-lab.org/
A so called thread-less event driven actor can suspend either the entire thread or just a continuation closure for an event. This approach combines the best of both worlds which means threads support blocking operations such as system I/O and can be executed on multiple processor cores in parallel. Event based actors is more lightweight and scales to large numbers of actors which leads to a significant higher message throughput compared to thread based actors [123].

More precisely, when using thread-based actors the throughput stagnates around 1.4 million messages per second whereas the same hardware\(^8\) can process up to 20 million messages per second using event based actors\(^8\)\(^4\). With a more sensitive matching of the number of actors to the actual number of CPU cores, the expected throughput can easily be doubled\(^8\)\(^5\). It is worth mentioning, the overall low throughput of thread based actors is mainly because of the high frequency of expensive context switches. The high throughput of event based actors is achieved by using delimited continuations for suspending and continuing tasks and adaptive work-stealing for distributing actor tasks dynamically on the fork-join framework [108].

Because of the unified actor model and the fairly sophisticated implementation of actors in Scala, the developer is shielded from all underlying concurrency complexity by a well documented high-level API [111]. Building a fully asynchronous, non-blocking actor system requires the following three steps:

A Message definitions for requests and replies

B Actor(s) for processing messages

C An actor system for organizing all actors

---

\(^8\)\(^3\) X 6 core AMD Opteron = 48 cores with 128 GB DDR3 memory

\(^8\)\(^4\) http://letitcrash.com/post/17607272336/scalability-of-fork-join-pool

\(^8\)\(^5\) http://letitcrash.com/post/20397701710/50-million-messages-per-second-on-a-single-machine
A: Message definitions

Actor message can be defined as an object but must be immutable in order to prevent all issues caused by mutable state and data. The recommended and commonly used approach is to use Scala case classes (see 4.3.6) which are immutable by default (5.1.4 in [111]). Furthermore, it is good practice to define case classes by extending an (abstract) trait which defines the category (i.e. reply) a message type belong to. This becomes more important with the upcoming support of typed communications channels finally solving the Type Pollution Problem\(^86\). The term type pollution was coined by Prof. Philip Wadler and describes the loss of type safety by handling multiple disjoint sets of types (6 in [124]). In Scala, this is the case whenever multiple communications message types are defined for one actor, for instance one types for processing messages and another types for receiving operational commands. The actual problem is that the type checker refers to the least common super-type between all messages which in Scala is Any and essentially removes all type safety. The solution would be to add a type constraint to a communication channel defining super-types per category of messages for each channel. As many typed input and output channels can be defined per actor, this allows the type safe handling of disjoint sets of message types.

In practice, this is exactly the reason why an (abstract)\(^87\) trait is required to define different categories of messages which then gets checked during compile-time against the type required for the communication channel. Listing 4.67 illustrates an example\(^88\) set of message types defined as case classes.

```scala
trait Request

case class Command(msg: String) extends Request
```

Listing 4.67: Message types for actors

\(^86\)http://doc.akka.io/docs/akka/2.2-M3/scala/typed-channels.html

\(^87\)the trait does not have to be abstract but it prevents accidental misuse.

\(^88\)4 in Munish2012
B: Actors

There are two different sub-types of actors in Scala, "normal" actors and "typed" actors. The only practical difference between these two sub-types is that a "normal" actor is written in just one single file whereas a typed actor has a public interface and a private implementation of the interface. Calls to the publicly defined interface are delegated asynchronously to the private instance of the implementation. The only reason for using typed actors is because of Java POJO’s\(^{89}\) can be represented as an actor which simplifies integration of existing systems.

Essentially, typed actors are the bridge between the object-oriented world and the asynchronous, message based actor world (4 in [10]). Furthermore, there are two different ways of communication between actors: Fire and forget and send and receive. The first one is a one way unidirectional message which is used for notifications whereas the second one is bi-directional communication for exchanging messages between actors. Figure 4.7 illustrates the difference in message passing between actors.

![Fire and Forget vs Send and Receive](image)

**Figure 4.7:** Synchronous and asynchronous actor messages (p. 84 in [10]).

The send and receive message sending is denoted as `?` or `ask` in Scala and can be handled asynchronously or synchronously. The former is non-blocking by using futures for a reply and the latter is fully blocking the working thread until an answer message has been received by the caller.

\(^{89}\)POJO: Plain Old Java Objects
Fire and forget on the other hand is denoted as ! or tell and just sends a message asynchronously one way. Listing 4.68 shows the syntax used for using fire and forget message sending. Listing 4.69 illustrates the usage of asynchronous send and receive which follows syntactically the same rules as tell but must be wrapped into a future for asynchronous processing (3 in [10])(5.1.5 in [111]).

Synchronous message processing on the other hand is only possible on typed actors by calling methods of the implementation directly. However, this is not advised as thread-blocking and context switching is expensive and reduces overall throughput. This is not even required for synchronous IO operations that usually would require thread blocking. There are specific monadic IO patterns available [98] which wraps synchronous IO operations into asynchronous futures enforcing strictly linear ordering.

Actors for data processing can be created in several different ways such as:

- Creating actors with default constructor
- Creating actors with non-default constructor
- Creating actors using Props and the actorOf factory method
- Creating actors using anonymous classes
- Creating actors using the Actor DSL

Each of these methods has a particular use case which is covered in detail in the reference manual (5.1 in [111]).
Chapter 4. Scala

For the scope of this thesis, only the default constructor and the `actorOf` factory method is used. The factory method is used to either create actors from within another actor or by creating an actor system. Creating a new actor (with or without default constructor) requires extending the abstract `Actor` class and implementing the receive method. A simple echo server example is used for illustrating the actor programming model. The server, as shown in listing 4.70 extends the abstract `Actor` class and defines the `receive` method as a fire and forget reply by replying the message with a comment back to the original sender. The important observation is that `Sender` is used to retrieve implicitly a reference of the original sender.

```scala
class ServerActor extends Actor {

  override def receive: Receive = {
    case message: String =>
      // Get reference to the message sender and reply back
      sender ! reply(message)
    case _ => println("Error: Unknown message type")
  }

  def reply(String s) = (message + " got something")
}
```

Listing 4.70: Server actor in Scala

The corresponding client actor is shown in listing 4.71 which sends any received message to the server and logs the result. For implementing a non-blocking, asynchronous message processing, futures are used to wait for the server response. The receive method is implemented using pattern matching (see 4.2.1) in order to handle different kinds of messages. The first pattern is the required type ”message” which is passed to the reply method which returns the original message with and attached string to it. The second case for any other kind of message, denoted with the underscore, simply prints an error to the system console and omits the actual message. Using this mechanism, independent from the actual error handling, means only correct messages are processed regardless of the occurrence of incorrect ones. Also, using pattern matching allows to process different message types with different methods. Finally, a time-out of five second is defined to detect slow responses. The example client does not handle any error out but in general actors can, for instance, re-send message in case of a time-out before propagating an error.
The server actor needs to be known to the client which can be either by URI as shown in the example or by loading a configuration file containing URI’s. The `actorFor` factory method is used to create a so called remote actor instance. Because of the URI address handling and implicit reference creation, Scala handles local and remote actors in the same way.

```scala
class ClientActor extends Actor
  with ActorLogging {

  val timeout = Timeout(5 seconds)
  val addr = "akka://ServerSys@localhost:2552/user/serverActor"
  var serverActor = context.actorFor(addr)

  override def receive: Receive = {
    case message: String =>
      val future = (serverActor ? message).mapTo[String]
      val result = Await.result(future, timeout.duration)
      log.info("Message received from Server -> ", result)
  }
}
```

Listing 4.71: Client actor in Scala

In order to make the client server actor example work, an actor system is required to create and manage actors.

C: Actor systems

An actor system is a collaborating ensemble of actors in the sense of organizing actors in one unit to manage shared facilities like scheduling services, configurations, logging, external resources etc. Several actor systems with different configuration may co-exist within the same JVM as there is nothing shared between systems and communication between systems is only based on immutable messages between actors. Thus, actor systems can and should be used as building blocks in a functional hierarchy. The most important property of an actor system is that there is no difference between local and remote actors. Scala actors using an URI identifier for each actor which means only by reading an actor URI one can determine if an actor is locally or remotely deployed. Conceptually and programming wise, all actors are handled exactly the same way regardless of the actual deployment.
An actor system defines scheduling mechanisms, message dispatching, message routing and supervision strategies. Beginning with scheduling, each actor system has one unique instance of a scheduler which delegates tasks to the default message dispatcher. The default implementation of Scheduler is based on the Netty HashedWheelTimer\(^90\). The default scheduler does not execute tasks at the exact time, but on every tick, it will run everything that is overdue. For specific needs, the tick time can be customized as required (5.5 in [111]).

A message dispatcher is what actually executes each actor task in an actor system. All MessageDispatcher implementations are also an ExecutionContext, which means that they can be used to execute arbitrary code on an execution context.

The default message dispatcher is a Dispatcher with a "fork-join-executor" that will be used in case nothing else is configured for an actor system. For customization, a different execution context, the minimum and maximum number of threads and a so called parallel factor can be configured independently for each actor system. The parallel factor is a multiplier on the number of available processors\(^91\) which is set to 2.0 by default. The value two is chosen for supporting HyperThreading\(^92\) of modern CPU’s which an increase performance by 12 %\(^93\). Event though fine tuning the dispatcher can optimize the throughput of an actor system, for general purpose this is often not required as the standard dispatcher already provide excellent throughout by default (5.9 in [111]).

Message routing is required to deliver messages to the right actor by using a router. A Router is an actor that routes incoming messages to outbound actors. An actor system can be run without a router, a custom router class or by using one of the pre-defined routers. Pre-defined message routers are:

- RoundRobinRouter
- RandomRouter
- SmallestMailboxRouter
- BroadcastRouter

---

\(^90\) [http://docs.jboss.org/netty/3.1/api/org/jboss/netty/util/HashedWheelTimer.html](http://docs.jboss.org/netty/3.1/api/org/jboss/netty/util/HashedWheelTimer.html)

\(^91\) That is the number of total cores in a system

\(^92\) HyperThreading is essentially a two way instruction pipeline for each physical core in order to fully utilize Arithmetic Logic Units(ALU)

• ScatterGatherFirstCompletedRouter
• ConsistentHashingRouter

Creating a router for an actor system only requires one or more actors that shall be served, the choice of router and the number of instance of the target actors. Alternatively, a list of already created actors can be used as well. Listing 4.72 illustrates these two choices. The first example in the listing passes the reference to the ExampleActor1 as argument to the router constructor for creating a round robin router that serves five instances of the router. The second example uses a vector of existing actor instances to create a round robin router to serve the three instance of ExampleActor1.

```scala
// 1) Create five instance of ExampleActor1 and use RR-Router
val router1 = system.actorOf(Props[ExampleActor1].withRouter(RoundRobinRouter(nrOfInstances = 5)))

// Use existing instance and a RR-Router
val actor1, actor2, actor3 = system.actorOf(Props[ExampleActor1])
val routes = Vector[ActorRef](actor1, actor2, actor3)
val router2 = system.actorOf(Props().withRouter(RoundRobinRouter(routes)))
```

Listing 4.72: Router for actor system

Using a router on several instance of the same actor can significantly improve the throughput of an actor system depending on the chosen routing strategy and number of instance. Optimizing message routing can increases the overall system performance significantly more compared to changing the underlying execution context thus careful measurement is mandatory to support a good routing configuration (5.9 in [111]). The core concept of an actor system is that tasks and responsibilities are split up and delegated until they become small enough to be handled in one piece by an actor. Consequently, each actor can reason about a small sub-task and decide if an error occurred and how to handle such a case. Conversely, many results of smaller tasks are aggregated into bigger ones using actors in a ”higher” position within the actor hierarchy. Those ”parent” actors deal with errors on a broader level.
Chapter 4. Scala

This hierarchical design of actor systems leads to error propagation means each actor only handles those errors the actor is defined for and escalates all other errors to the next higher actor to deal with. This recursive structure then allows dealing with errors on the appropriate level. Designing an actor system requires making decisions about which actor should supervise what kind of task (2.2.1 in [111]). A supervisor actor delegates tasks to subordinates and therefore must respond to their failures. For instance, when a subordinate detects a failure (i.e. exception), it suspends itself and all its subordinates and sends a message to its supervisor, signaling failure. Depending on the task and the exact error, the supervisor actor has a choice of the following four options (2.4 in [111]):

- Resume the subordinate, keeping its accumulated internal state
- Restart the subordinate, clearing out its accumulated internal state
- Terminate the subordinate permanently
- Escalate the failure

The fourth option is rooted in the core concept of splitting tasks and responsibilities and handling them at the right level within an actor system which leads to a recursive fault handling structure. Scala actors implements a specific form called "parental supervision" which means actors can only be created by other actors and each created actor is supervised by its parent. This concept forms an implicit supervision hierarchies by default which can be customized by adding error handling (or escalating) to an actor as required. Parental supervising prevents that actors cannot be orphaned or attached to supervisors from the outside, which might otherwise catch them unawares. Finally, having an implicit supervision hierarchy means an exact order required for controlled system shut-down is present which yields clean shut-down procedures for any sub-trees of an actor system (2.4 in [111]).

Building a first actor system in practice does not require to customize scheduling, dispatching, routing and supervision strategies. The default values are usually sufficient for common scenarios but whenever error handling becomes relevant or system throughput needs to be optimized any of those core concepts is mandatory to be understood before any appropriate changes can be performed. Using the client server

---

94The top-level actor is provided by the library by default
example from listing 4.71 and listing 4.70 a very basic actor system only requires an
instance of an \textit{ActorSystem} and at least one instance of one actor.
The example\footnote{\url{http://goo.gl/IAOFR}} in listing 4.73 illustrates a minimal actor system that starts the server
actor to be ready receiving messages from a client. The configuration file is skipped
in the example but essentially it contains only host-name, port and providing class
to make sure the server actor servers as remote actor to the outside world. The name
of the actor system is a system wide identifier and must be unique.

```scala
object ServerActorSystem {
  def main(args: Array[String]): Unit = {
    // load the configuration
    val config = ConfigFactory.load().getConfig("ServerSys")
    val system = ActorSystem("ServerSys", config)
    val serverActor = system.actorOf(Props[ServerActor], name = "serverActor")
  }
}
```

\textbf{Listing 4.73: Basic actor system in Scala}

The name of the actor must be only unique within the scope of the actor system.
A different actor system on the same JVM may have another actor with the same
name. This is not an issue because Scala actors are looked up by reference which
always depends on the actual actor system (2.5 in [111]).

Creating an actor system for the client actor follows the same principle of creating
an instance of an ActorSystem and at least one actor. Listing 4.74 shows a minimal
actor system required to start the client actor. First, the configuration needs to be
loaded. This step is only required for dealing with remote actors. Having all actors
local does not require any configuration. The second step is creating a client actor
system using the retrieved configuration. Third, an instance of the client actor from
listing 4.71 is created. Communication with the server actor is asynchronous by cre-
ating a future (see 4.11.5) and by waiting non-blocking on the result (5) for a given
duration time of five seconds. Next, the result is stored in a field \textit{(result)} and and
printed (6) to the system console. Finally, in order to terminate an actor system
correctly, the system shutdown method (7) needs to be called.
object ClientActorSystem {
  def main(args: Array[String]): Unit = {
    val timeout = 5 Seconds
    val config = ConfigFactory.load().getConfig("ClientSys")
    val system = ActorSystem("WorkerSys", config)
    val clientActor = system.actorOf(Props[ClientActor], name = "clientActor")
    val future = (system ? "Hello Server").mapTo[String]
    val result = Await.result(future, timeout.duration)
    println("Message received from Server -> ", result)
    system.shutdown()
  }
}

Listing 4.74: Client actor system in Scala

Actor systems enable the composition of local and remote actors to build highly available and error resistant real-time data processing systems in Scala. However, this comes at the price of increased complexity. As the simple echo server already demonstrates it requires four classes to be written in order to perform the most rudimentary client server communication using actors. Adding exception handling, error propagation, supervision strategies and routing to the example will significantly increase the overall complexity of the code. Despite all usefulness and well thought concepts, building actor systems requires in-depth knowledge and skills because of the inherit high level of complexity.
4.11.7 Disruptor

The disruptor framework is actively developed by LMAX ltd. as foundation of their high performance and low latency stock trading platform. According to a review from Martin Fowler, the disruptor can handle 6 million orders per second on a single thread.

In order to accomplish high system throughput while preserving low latency, an internal investigation of JVM concurrency has exposed that many applications depend on queues to exchange data between processing stages. Queues, however, are responsible for very high latency costs mainly caused by contention on CPU cache level. Preventing contention on CPU level requires a different approach which is entirely designed around the goal to match concurrency data processing to modern CPU architecture in order to accomplish the highest possible throughput on a CPU. This considerations has led to the disruptor framework which is based on the concept of non-blocking event processing using a bounded, pre-allocated, circular buffer as main structure to exchange data concurrently.

The ring-buffer is pre-allocated at start-up time and can store either an array of pointers to entries or an array of structures representing the entries. Because of a limitation of the Java language, entries associated with the ring-buffer are represented as pointers to objects. Each of these entries is typically not the data being passed itself, but a container for it.

The critical point is, if the last reference is lost, entries will be marked for garbage collection which can happen at any time. Garbage collection (GC) on the JVM is an expensive operation, at least on the original Sun HotSpot JVM widely used, which decreases latency and throughput noticeably. Essentially, the more memory is allocated the greater is the burden of unintended garbage collection.

On the other hand, the pre-allocation of entries in a ring-buffer eliminates this issue and prevents garbage collection since entries will be re-used and live for the entire duration of the Disruptor instance.

---

96 http://www.lmax.com/
97 LMAX claims 400 million orders/day
98 LMAX claims a 99 percentile latency of under 3 milliseconds
99 http://martinfowler.com/articles/lmax.html
100 The name Disruptor is an analogy to the Phaser introduced in JDK 7 to support fork-join
101 It needs to be said, Zing JVM is a GC free JVM alternative
Chapter 4. Scala

The next important difference to existing queue based systems is a fine-grained separation of concerns implemented in the disruptor. Three main tasks are separated in the disruptor design:

- Storage of items being exchanged through a ring buffer
- Coordination of producers pushing a new item to a ring buffer
- Coordination of consumers being notified that a new item can be pulled from a ring buffer

The disruptor uses sequencing as its central concept of concurrency which means all elements have a unique sequence number which is used for reading and writing operations on the ring-buffer. This sequence of the next available slot can be a simple counter in the case of only one producer or an atomic counter updated using Compare And Swap (CAS) operations in the case of multiple producers.

The important difference is the claiming strategy which inherently coupled to the used data-structure. Each event has a unique sequence number which this sequence number can be traversed sequentially for all published events. For clarification, a ring structure means, if the last entry of a ring buffer is reached, the next entry is the first entry which allows interruption reading. Because of this difference, traversing data is very friendly to the caching strategies employed by modern processors avoiding expensive cache misses. Cache misses are absent because once the entire ring-buffer is stored in a CPU cache\textsuperscript{102}, any sequence number at any position is claimed in constant(!) time. Furthermore, because of the ring structure, several events can be read in a batch which allows interruption free data processing.

Producers are pushing items to a ring buffer in a two phase commit. In a first phase, a unique sequence number for a free slot is claimed. Once such a unique sequence number is obtained, a producer has exclusive write access to that free slot in the ring buffer. Because of this exclusive write access to a slot, a CAS operation for claiming a sequence number is required for multiple producers in order to prevent conflicting writing. In a second phase, the producer writes a new event to the ring buffer and returns the claimed sequence number. This two phase commit is also called publishing an event.

\textsuperscript{102}The main goal of a ring buffer is to fit at least in the L3 Cache of a CPU
When the producer has finished updating the entry it finalizes the commit by updating a cursor on the ring buffer for the latest entry available to consumers. The ring buffer cursor can be read in a busy spin by the producers using memory barrier without requiring a CAS operation. This is important because CAS implies a certain delay to ensure the most up to date version is returned. Reading a cursor without CAS is faster because this delay is removed.

Reading from the ring buffer is coordinate by the cursor indicating the highest slot available for reading. Consumers wait for a given sequence to become available by using a memory barrier to read the cursor. Once the cursor has been updated the memory barriers ensure the changes to the entries in the ring buffer are visible to the consumers who have waited on the cursor advancing.

Each consumer contains their own (internal) sequence which they update as they process entries from the ring buffer. Consumer sequences allow the producers to track consumers to prevent the ring from wrapping. Consumer sequences also allow coordinate work on the same entry in an ordered manner. In the case of having only one producer, regardless of how many consumers are used, neither locks nor CAS operations are required for concurrent event consumption since all concurrency coordination is done with memory barriers on sequences.

There are three important details in the disruptor implementation (4 in [17]). First, on most processors there is a very high cost for the remainder calculation on the sequence number, which determines the slot in the ring. This cost is minimized by making the ring size a power of two. A bit mask of size minus one can be used to perform the remainder operation very fast and efficiently.

Second, because of this convention, a consumer can perform *batch reading* which means, if several slots are available, all can be read at once instead of reading one entry after another. This prevents CPU caches misses and increases overall performance (4.4 in [17]).

Third, the consumer sequence allows multiple consumers to read the same entry simultaneously. For instance, if an event is published to the ring-buffer, three consumers can read the entry simultaneously and perform several tasks such as logging, back-up and processing. All these tasks are executed in parallel thus does not affect system latency or throughput.
Programming the disruptor in practice requires only a few steps:

- An event JavaBean with a static `EventFactory`
- An `EventHandler` for processing events
- A `RingBuffer` that is used for publish events
- An consumer for processing events

Listing 4.75 shows a simple Java bean required for the disruptor. The only noticeable difference is the implementation of `EventFactory` using an anonymous inner class. This is only due to the convention of the disruptor using internally an abstract factory to create beans; thus each bean requires a type specific implementation of the abstract factory.

```java
public final class ValueEvent {
    private long value;

    public long getValue(){return value;}

    public void setValue(final long value) {this.value = value;}

    public final static EventFactory<ValueEvent> EVENT_FACTORY = new EventFactory<ValueEvent>() {
        @Override public ValueEvent newInstance() {
            return new ValueEvent();
        }
    };
}
```

**Listing 4.75**: Disruptor event in Java

The usage of normal Java Beans in a concurrency framework may surprise but mutable data structures are not an issue because a producer always has exclusive write access to a free slot. In practice, a ring-buffer has a number of pre-allocated events, which are initially empty and once a free slot is claimed, the producer calls the setter of an event is called to write the content exclusively into the slot. In case a claimed slot already contains some data, the event is simply overridden by calling the setter again. This is also thread safe because of the exclusive write access each producer has on a claimed slot.
Listing 4.76 illustrates the usage of the ring-buffer for event processing. First a new standard Java execution service needs to be created. Next, a new instance of the disruptor is created using ValueEvent as event type to process, the previously created execution service and a specific size for pre-allocation of the ring-buffer.

```java
public class Simple {
    private static final int _NR_CORES = 2;
    private static final int _SIZE = 1024;

    public static void main(String[] args) {
        ExecutorService exec = Executors.newFixedThreadPool(_NR_CORES); // (1)
        Disruptor<ValueEvent> disruptor = new Disruptor<>\(ValueEvent.EVENT_FACTORY, _SIZE, exec\); // (2)

        final EventHandler<ValueEvent> handler = new EventHandler<ValueEvent>() {
            // (3)
            public void onEvent(final ValueEvent event, final long sequence, final boolean endOfBatch) throws Exception {
                System.out.println("[EventHandler] Sequence: "+ sequence);
                System.out.println("[EventHandler] ValueEvent: "+
                        event.getValue());
            }
        };

        disruptor.handleEventsWith(handler); // (4)
        RingBuffer<ValueEvent> ringBuffer = disruptor.start(); // (5)

        for (long i = 5; i <= 200; i++) {
            long seq = ringBuffer.next(); // (6)
            ValueEvent valueEvent = ringBuffer.getPreallocated(seq); // (7)
            valueEvent.setValue(i); // (8)
            ringBuffer.publish(seq); // (9)
        }
        disruptor.shutdown(); // (10)
        exec.shutdown();
    }
}
```

Listing 4.76: Disruptor ring buffer in Java

Next, an event handler is implemented using an anonymous inner class (3) to print out the sequence number and the value stored inside the event. Using an anonymous inner class is simplest way to create an event handler but in practice implementing the corresponding event handler interface is the preferred way.
Chapter 4. Scala

This event handler is next assigned to the disruptor to ensure events are processed correctly (4). Only after that step, a correct ring buffer can be created (5) and process events on publishing. The actual usage of a ring-buffer is rather trivial and only requires claiming the next sequence number (6), allocating a new value event from the claimed slot (7), updating the value of the event (8) and publish the event to the ring-buffer (9). The last step triggers the event handler to process the event. Finally, both the disruptor and the used execution service needs to be shut-down to free all allocated resources.

As the example already indicates, implementing concurrency using the disruptor is not trivial and requires an in-depth understanding of the underlying concepts. In particular, implementing all required factories, handlers and interfaces is not exactly a high level of productivity. Nevertheless, the very low latency and high throughput justifies the adoption of the disruptor whenever these requirements are mandatory. In summary, despite its complex concepts, the disruptor is currently (2013) the fastest known concurrency approach for Java. The results in chapter 7 show how concurrency in Scala compares to the disruptor in terms of performance.
Chapter 5

Results

“Anyone who has never made a mistake has never tried anything new. “

Albert Einstein, 1879 − 1955

The four main results of this thesis are based on the review of fifteen Java DBC projects and their limitations, as summarized in section 3.6. Based on the review, I have implemented a custom DBC tool, called f-DBC, in Scala that resolves most of the identified limitations. Implementation details and some examples are presented in chapter 6. In order to provide evidence of the usefulness of f-DBC, I did a case study for illustrating the advantages of the chosen approach. Also, part of the case study is an empirical performance evaluation showing that the usage of f-DBC only affects application run-time performance within the 0.1% tolerance of measurements. Details of the case study are summarized in chapter 7.

Using contracts as test-oracles for test automation only assumes a written contract that can be transformed into a test oracle which determines if a generated test has been passed or failed. With f-DBC those contracts can be written thus the transformation and test generation would be a matter of tool support. The underlying assumption is that developers are writing contracts that can be used. I have asked the question: if tool support would not an be issue, how likely would developers write contracts as required for automated testing?

In order to answer this question, I did an empirical data analysis of DBC usage in nine Eiffel projects. Apart from Eiffel studio IDE, contracts are barely used at all.
Chapter 5. Results

The next question was: Do Scala developers writing contracts?
I did an empirical evaluation on twelve Scala projects with the results that Scala developers write even less contracts compared to Eiffel developers. In particular, post-conditions are not written at all in Scala. Detailed results and methodology of the empirical evaluation are summarized in chapter 8.
Inspired by these results, I’ve drawn the conclusion that the assumption of written contracts is not verifiable in practice thus a different approach is required. Next, I was asking the question: Is there another way of verifying correctness of functional concurrency programming in Scala?
One answer to this question is the usage of a compiler macro that prevents commonly known pitfalls such as impure functions, mutable fields or null references. Even though this is a considerable small contribution, compiler supported detection of side effects has turned out to reduce debugging time noticeable.
The results are organized in the three following chapters. Chapter 6 covers f-DBC in detail. Chapter 7 presents a case study of f-DBC applied to different concurrency programming in Scala. Chapter 8 covers the empirical DBC usage study and the compiler macro is presented in chapter 9.
Chapter 6

f-DBC

6.1 Overview

Based on the review of existing Design by Contract tools, I’ve drawn the conclusion that none of the existing tools is flexible yet powerful enough to formulate contracts that can be used as test oracles. The lack of expression power is due to the lack of predicate support and the lack of flexibility is caused by rather limited possibilities to configure how to enforce contracts. The analysis of existing tools is summarized in section 3.6.

Based on the gained insights, I’ve designed and implemented an entirely new approach of contracting called "functional design by contract" (f-DBC). The full source code of f-DBC is available online\(^1\).

The used architecture applied to f-DBC is explained in section 6.4. The cake pattern (see section 4.8) based implementation of f-DBC is presented in section 6.6. The usage of f-DBC is illustrated on an example in section 6.2 and finally a comparison to Scala’s default DBC is made in section 6.3.

I’ve chosen Scala to implement f-DBC as there are many reasons that favours Scala over Java such as Scala’s full compatibility to Java while offering functional and object oriented programming on one language. Chapter 4 provides more details about Scala.

\(^1\)Available at: [http://marvin-hansen.github.io/Msc/](http://marvin-hansen.github.io/Msc/)
f-DBC is different from all other existing DBC implementations by providing a broad variety of new functionality unified in one single tool.

1. Functional programming support

2. Flexible contract enforcement

3. Profile based configuration

4. Concurrency support

5. Logging support

6. Flexible handling of contract violation

7. Extensible component model allowing customization

By support for functional programming, I refer to passing functions as arguments to pre or postconditions which then will be evaluated into a boolean value determining if the evaluation of the function was successful or not. This means, custom functions for complex evaluations are supported which is similar to dynamic predicates in Ada (see 3.5).

By profiles, I refer to a set of configurations that are specific to a certain use case such as testing or production. Each profile specifies how contract violations should be handled, if or what kind of logging should be used and how contraction violations are reported. By default, f-DBC comes with four pre-configured profiles for development, debugging, testing and production.

Aliases take the profile approach on the next level by allowing an alias to be linked to a profile. This means, an alias can be defined for each module, package, or even one single class. Having one alias per module allows swapping DBC profiles independently at different stages for each module.

Concurrency is supported in f-DBC by a concurrent contract checker which means large contracts containing many expressions are evaluated concurrently.

A flexible logging facility is already part of f-DBC so that a generic interface allows to plug-in the actual logging framework. Through that, contract checks can be logged at different levels, for instance, all checks, only passed checks or only failed checks can be logged in f-DBC.
Contract violations can be handled in a flexible way. One profile may use email notification, another profile may log the violation and another one may terminate the application. Profile based handling enables handling contract violations differently depending on the actual requirements for each part of a project. Even more, profiles can be applied on a single class allowing specific monitoring of a particular problematic class.

Finally, f-DBC uses an extensible component model that allows the inclusion of custom components. In case a specific notification or handling mechanism is required, a custom component can implement any of the existing interfaces and used in a custom profile. Through aliases, this new custom profile can be instantly connected to existing contracts without any code re-factoring. Even inheriting and extending existing interfaces is supported in case none of the default solutions provided by f-DBC are covering the actual requirement.

Instead of forcing convention that may not fit into each project, f-DBC can be customized down to the smallest detail so that f-DBC fits the actual project. Through this open extension model, there are no arbitrary limitations which means, f-DBC can evolve over time even beyond contracts.

6.2 Example

To illustrate the level of flexibility offered by f-DBC, a small usage example is used to clarify the main difference to existing work further. Beginning with a small example, an account class is used to define a contract to guard certain assumptions such as a correct opening balance. Let’s consider the account example as part of a larger project for building a system for a financial service provider. In this case, the project is developed in different teams all supervised by one central quality assurance (QA) team. Over time, the QA team observed that not all teams provide the same level of code quality which means testing effort has grown significantly. In order to reduce the number of but detecting tests, the QA team provides skeleton interfaces, contracted in f-DBC which then gets implemented by different teams. In order to take full advantage of the flexible configuration offers, one central alias file is used. Listing 6.22 shows a complete listing that creates a set of aliases linked to specific profiles. Each profile is defined as trait and a companion object.
A trait can be mixed into any other trait or class whereas a companion is used for imports. The latter is particular important to preserve full compatibility with Java. The first alias accDBC is used for the contracting the account module and is linked to the development profile. As the new systems adopts the MVC\(^2\) pattern, the remaining three aliases are used to contract the user interface, the business logic and persistence layer. One observation is that GuiDBC, the alias for the user interface is linked to a mock profile, which essentially does nothing. During early GUI development, there is usually a certain level of experimentation required thus using a mock profile does allow writing contracts but the lack of enforcement does not interfere with the fast changing nature of experimenting. Once a more stable level is reached, the mock profile can be swapped for a logging profile that only logs contract violations to make sure the user interface can be tested interruption free while collecting reports of failed contracts. Back to the account example, using f-DBC in practice only requires two

```scala
// (1) account module alias
object accDBC extends fDBC
trait accDBC extends DevelopmentProfile
// (2) GUI specific alias
object GuiDBC extends GuiDBC
trait GuiDBC extends MockProfile
// (3) alias for business logic
object ModelDBC extends ModelDBC
trait ModelDBC extends TestingProfile
// (4) Specific alias for data layer
object PersistenceDBC extends PersistenceDBC
trait PersistenceDBC extends DevelopmentProfile
```

Listing 6.1: f-DBC aliases

steps. First, the companion objects corresponding to the chose alias needs to be imported (1). Alternatively, the alias trait can be mixed in (2). Second, contracts needs to be written. Using the account example, already known from section 3.4, listing 6.2 illustrates how to write contracts in f-DBC. All methods accessible through an alias require a dbc prefix which means there is no static reference to any implementation, composition or profile which means whatever implementation or profile may change, the code in the account example is not affected.

\(^2\)Model, View Controller
Chapter 6. f-DBC

Listing 6.2: f-DBC: Writing contracts

```scala
// (1)
import profiles.accDBC dbc

class Account_fDBC extends AccountI with accDBC {
  // (2) alternative usage
  var accBalance = 0
  val nonDebitableAcc = false

  override def openAccount(b: Int) {
    // (3)
    dbc.precondition("AccountC4S opening balance must be zero or greater", b >= 0)
    accBalance = b
    // (4)
    dbc.postcondition("Ensuring balance has been set correctly", accBalance.equals(b))
  }
  ...
}
```

For instance, swapping the development profile used for the accDBC alias, for a debugging profile only means, a contract violation does not terminate the application any more but instead will be printed to the system console. Once the account class has been fully debugged and tested (using the testing profile) it can be deployed in production using a production profile. A production profile can use, for instance a silent alert to notify the system operator of a contract violation.

Three observations are important. First, staging the account example from development to any other stage does not affect the contract enforcement of any other module, package or class. Using aliases means, each alias uses their own profile so that any other module of the same project can be staged independently.

Second, the lack of static references to profiles and compositions allow the complete replacement of any existing f-DBC component at any time. If the project grows and requirement changes, f-DBC can be changed to meet the new requirements. For instance, let’s assume the project has deployed a logging server to centralize logging from all testing systems. In f-DBC, this means, one new implementation for the logging component needs to be written. As profiles and aliases are fully decoupled, only the affected profiles needs to be updated to ensure the new logging is used project wide amongst all aliases.
Chapter 6. f-DBC

Third, even if the example uses one central class to manage all aliases, this is not necessary. Instead, many different ways can be used, for instance, for small projects, aliases and profiles can be managed in one central file. For very large and distributed projects, template files can be used and shared but still allowing team specific customizations.

6.3 Comparison

A direct comparison between fDBC and Contracts for Scala$^3$ (C4S) shows a significant difference in lines of codes (LoC) required writing contracts. Lines of code is measured as following: All comments have been removed thus only code is measured and blank lines are included. All code was automatically formatted according to the IDE$^4$ default coding guidelines which are equivalent to the Scala coding style. A metric plugin was used to count LoC for each version of each example. Furthermore, the editor’s built-in ”view line number” function was enabled to verify the number reported by the metric plugin.

For the comparison, four examples from undergraduate programming exercises have been used to write example contracts. The base version is the program written in plain Scala without any contracts. The C4S version follows the guidelines from ”Contract for Scala” as presented by Odersky [61]. The fDBC version uses the $fDBC$ default alias without any modification. Table 6.1 shows the actual LoC for each example.

<table>
<thead>
<tr>
<th>Example</th>
<th>Base</th>
<th>C4S</th>
<th>fDBC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Account</td>
<td>28</td>
<td>76</td>
<td>40</td>
</tr>
<tr>
<td>Circle</td>
<td>12</td>
<td>30</td>
<td>17</td>
</tr>
<tr>
<td>Coke</td>
<td>11</td>
<td>31</td>
<td>15</td>
</tr>
<tr>
<td>Pizza</td>
<td>12</td>
<td>30</td>
<td>16</td>
</tr>
</tbody>
</table>

Table 6.1: f-DBC: Comparison to C4S

For the circle, coke and pizza example, fDBC requires just 4 more lines of code to cover the same functionality as C4S provides.

$^3$contracts for Scala is covered in section 3.4
$^4$InteliJ IDEA 12
C4S on the other hand requires 18 additional LoC’s, totalling four times more code. Considering the account example, which has four methods to contract, the difference becomes clearer. fDBC requires 12 more lines whereas C4S requires 48 more lines which is five times more code.

The observation is clear in terms of LoC, fDBC is four to five times more compact compared to C4S while offering significant more flexibility in terms of profiles and contract violation handling. Also, one shall note that C4S only allow one single boolean expression per contract which means, checking three preconditions requires three lines of code whereas fDBC can check any arbitrary number of boolean expressions in a precondition.

The next sections provide more details of the underlying architecture and component composition.

### 6.4 Architecture

A first version used modular mixing composition (see 4.7) but this has turned out to have insufficient dependency management (see 4.8.2 for details). In particular, the case when one component is required to implement another without introducing static references between those components is rather insufficiently covered. Instead, I’ve used the cake pattern (see 4.8) to implement f-DBC as a component based systems. Essentially, f-DBC is based on only three core components:

1. Checker
2. Handler
3. Logger

The checker component defines all core design by contract functionality, such as precondition, postcondition and invariant checks. In addition to that, there are are conditional checks and and equality checks. Each of those methods returns true or false indicating if a check has passed. The handler component defines just one method `handle` which has no return type (`Unit`). The handle method is only called in case of a failed contract check and requires further action for handling the contract violation.
Depending on the actual implementation, different actions can be performed such as terminating the application, printing to the system console, writing a log or sending a notification. Finally, the logger component is used to define a logger that should be used for logging contract violations. By default, f-DBC comes with an implementation of the logback logger but any other logger can be implemented as well. Even though these three components appear rather simplistic, the design was chosen by purpose in order to allow a thin layered architecture. Currently, there are three layers with each layer responsible for a separate concern:

1. Core component layer

2. Composition layer

3. Profile layer

The first layer is built of three core components: *CheckerComponent*, *HandlerComponent* and *LoggerComponent*. It is important to note, that each component has only one unique task specified and there are zero dependencies between these components which means each of the component can be re-used in any other context. Also, task isolation implies a high level of cohesion as there is no intersection in functionality. The second architecture layer is the integration layer which is responsible for component composition. These composition layer only exists to model abstract compositions such as ”logged checker with handler” by combining those component dependencies together. However, the composition layer does not specify any implementation. Instead, the third layer, the profile layer, uses only functionality from the integration layer and specifies the actual implementation that should be used for this functionality. For clarification, the composition layer defines a ”logged checker with handler” but the profile layer takes this combined functionality and defines, for instance, two different profiles. The next section explains the concept of profiles in more detail.
6.5 Concepts

Conceptually, the profile layer allows the re-usage of the same abstract composition in different profiles and the re-usage of the same profiles under different aliases. Let’s take the example of a composition ”logged checker with handler” which is re-used in two different profiles. The first profile could be a logback logger, with a linear checker and a termination handler. A linear checker performs all checks in strict sequential order and a termination handler terminates an application with and error in case of a contract violation.

The second profile could be an apache logger with a linear checker and a notification handler. Both profiles re-use the same composition but apply different implementations, depending on the actual requirements. This is possible because of f-DBC’s architecture of separating implementing components from composition. Essentially, a composition defines what to do, for instance handle and logging, but a profile defines how to do the tasks of a compositions. A profile connects a composition with implementing components and makes such a configuration accessible.

Each profile is accessible as a trait and exported as a companion object which means different profiles can be applied to different parts of a project either by using modular mix-in composition or simply importing the companion object.

Using ”alias-traits” that are traits only used as an alias for another traits allows central management for swapping profiles, depending on the project stage. For instance, an alias-trait for a data abstraction layer (DAL) can be DAL-DBC which first aliases to a development profile but during deployment the alias might be switched to a production profile allowing silent alerts in case of contract violations.

By implication, this means changing for instance a logger is as simple as writing a new logger implementation and changing just one keyword(!) in the affected profile. All aliases referring to that profile will use instantly the new logger. Likewise, creating a new profile is as simple as extending an existing profile interface but choosing different implementations to specify the behaviour of the new profile. Aliases only need to be switched to the new profile and the new profile will be used wherever the alias is used in the project.
In summary, the low level of maintenance is a direct consequence of the cake pattern in conjunction with a high level of cohesion of those three core components. This, three layer architecture is implemented in just six Scala classes containing all predefined implementations for each component. The next section provides more details about the actual implementation details.

6.6 Implementation

Implementing f-DBC takes full advantage of Scala’s cake pattern\(^5\). Essentially, each of the core components is a cake component having several implementations providing the actual functionality. The composition layer consists of cake components that compose the core components together. Profiles on the other hand are mixins of compositions and core components accessible as a trait and companion object. All core components in f-DBC follow the principle of minimalistic interface design. In order to preserve the ideal of one task per component, minimal interface design is driven by the question:

\[ \text{Is there anything unnecessary left that can be removed?} \]

If the answer to that question is no, the interface is minimal according to its defined task. The consequence of this minimalistic approach is broad variety of possible implementations since there are only very few constrains defined in the interface. All three core components already have several different implementations, as summarized in table 6.2:

\(^5\)Cake pattern is covered in section 4.8
6.6.1 Core components

Checker component

Beginning with the `CheckerComponent`, the component can be used either through the composition layer or as a stand-alone module. The later may surprise but as the component has no dependencies, there are no obstacles to re-use the same component in a different, even non cake pattern, context. Listing 6.3 shows the very top level declarations of the `CheckerComponent` that defines a companion object (1) and a public trait (2) for accessing a `CheckerModule`. A `CheckerModule` is the composition of the interface and a chosen implementation. The companion object is used for static access through `import` and the trait for mixing the handler into any other trait or class.

```
1 /** (1) companion object for imports */
2 object CheckerModule extends CheckerModule
3
4 /** (1) Chose your checker here, default checker performs sequential checks */
5 trait CheckerModule extends CheckerComponent with SequentialCheckerImpl
```

**Listing 6.3:** f-DBC Handler access options

As already indicated, the stand-alone usage of the components applies only for implementations that does not have any dependencies. This is the case for almost all provided default implementations with the exception of one particular handler.
This particularity is documented in the corresponding implementation to prevent unintended side effects.

The *CheckerComponent* interface is defined in listing 6.4. There are a few important observations. First, the `@NotNull` annotation enforces a null check of the compiler which means such a parameter cannot be null thus preventing null pointer exceptions during run-time.

Second, there is no specific invariant check. An invariant check\(^6\) checks whether a state has accidentally change after initialization or not. This is because of the Scala style guide discourage the usage of mutable fields\(^7\). Instead, the preferred way in Scala is mutating parameters and using only immutable fields which is a known best practices in functional programming. In order to emulate the behaviour of invariant checks, an *equal* method is defined in the interface allowing explicit assertion if a particular field contains a specific value. This makes it simpler to reason about the underlying assumption regarding the expected to be unchanged value of a field.

Third, there are no specific pre or postconditions signatures defined in the interface. Instead, a generic *check*, *conditionalCheck* and *eval* methods are used. This is because the actual difference between a pre and a postcondition is just a name and the associated convention where to place those checks. The actual naming of those check functions is part of the composition where a check method can have any name but always refers to the check method defined in the *CheckerComponent*. A *conditionalCheck* is used, whenever a check requires certain pre-condition to be true. For instance, this allows the usage of boolean flags to switch only certain checks on and off as required. Finally, the *eval* function is a higher ordered function\(^8\) that takes any function of any type that has boolean as return type. This allows the usage of functions to express any kind of constrain similar to dynamic predicates in ADA\(^9\).

\(^6\)For details on invariance see section 3.1.2
\(^7\)http://docs.scala-lang.org/style/naming-conventions.html
\(^8\)For details on higher ordered functions, see section: 4.9.2
\(^9\)For details about ADA, see section 3.5
Chapter 6. f-DBC

```scala
trait CheckerComponent {
  protected val checkerRepo: CheckerRepo

  protected trait CheckerRepo {
    def check(@NotNull b: List[Boolean]): Boolean
    def equal(@NotNull A: Any, @NotNull B: Any): Boolean
    def eval(@NotNull f: => Boolean): Boolean
    def conditionalCheck(@NotNull cond: Boolean,
                          @NotNull b: Array[Boolean]): Boolean
  }
}
```

**Listing 6.4:** f-DBC Checker interface

The `CheckerComponent` has two implementations. The fist one performs sequential checks of all passed boolean expressions and the second one allows parallel evaluation of all checks. The sequential checker implementation is shown in listing 6.5. A few important observations. First, the check method is implemented using a list of boolean expressions which uses the `forall` method of Scala’s collection to ensure all required expressions to be true. Second, the equal function refers to the generic `equals` method which is inherited through `Scala.Object` to any class by default. Third, the return type of `eval` is the return type of the passed function which is the returning boolean value. More precisely, `eval` is actually an identity function limited to type boolean. Fourth, conditional check requires the condition and the actual expression to be true. The checking order implies, if the pre-condition fails, the remaining expressions will not be evaluated.

Finally, the difference between the sequential checker and the parallel checker implementation is only the usage of parallel `forall` within the `check` and `conditionalCheck` method. Using parallel `forall` is possible because of Scala’s unified collection\(^{10}\) which means, swapping a sequential boolean list for a parallel boolean requires only a `b.par.forall` inside the method body. Apart from this particular minor change, the parallel checker is identical to the sequential one thus the implementation details are omitted here\(^{11}\). Once a check has been performed, the result is handled through the handler component.

---

\(^{10}\)For details on parallel collection, see section: 4.11.4

\(^{11}\)The interested reader can verify this claim by reviewing the source which is accessible through the companion website: [http://marvin-hansen.github.io/Msc/](http://marvin-hansen.github.io/Msc/)
Chapter 6. f-DBC

```scala
trait SequentialCheckerImpl extends CheckerComponent {
  override protected val checkerRepo : CheckerRepo = new DefaultCheckerReporImpl

  private[this] class DefaultCheckerReporImpl extends CheckerRepo {
    override def check(b : List[Boolean]) : Boolean = b.forall(_ == true)
    override def equal(A: Any, B: Any) : Boolean = { A.equals(B) }
    override def eval(f : => Boolean) : Boolean = { f }
    override def conditionalCheck(cond : Boolean, b : Array[Boolean]) : Boolean = { cond && b.forall(_ == true) }
  }
}
```

Listing 6.5: f-DBC Checker implementation

**Handler component**

The interface of the `HandlerComponent` is defined in listing 6.6. There are three details worth further clarification. First, the abstract type member\(^\text{12}\) is of type `HandlerRepo` which is defined in the nested protected trait `HandlerRepo(2)`.

```scala
trait HandlerComponent {
  protected val handlerRepo : HandlerRepo // (1)
  protected trait HandlerRepo { // (2)
    def handle(@NotNull b : Boolean, @NotNull s : String) : Unit // (3)
  }
}
```

Listing 6.6: f-DBC Handler interface

Second, the `HandlerRepo` trait only defines one single method, by purpose. The concept of a handler is to be called to handle any contract violation thus only one method is required to do so. If more a more complex violation handling is required, a custom handler can be used instead. Third, the `handle` method is of type Unit (void in Java) because this does not cast the need to handle return values. This may sound trivial but handling return values of a handler heavily depends on the

\(^{12}\)Abstract type members are used to avoid static references between components. See section 4.8.5 for details
actual implementation and thus is better located in a custom handler component. For generic cases, and the example implementations illustrate those scenarios, there is no need to assume a handler may fail in handling a contract violation.

Taking the exit handler implementation from listing 6.7 as a first example, the benefit of cake component and interface minimalism becomes obvious. First, while implementing the handle method (2), inside a nested private class allows a very simple implementation swapping. It cannot be assumed that \texttt{System.exit} (3) is the best way to shut-down all kinds of applications but as the target application is not known, the exit handler can already be used.

In case an application requires a different shut down sequence, the \textit{same} handler can be used but only the internal implementation needs to be replaced. In practice, a controlled shutdown may consist of calling connection factory from an application server for closing all connection of the application before calling the server to shut-down the application. As this process depends on the actual application and used server, a custom nested private class needs to be written.

Next, the abstract type member(X) needs to be overridden by using the custom shut-down class instead of the default exit handler implementation. The important difference here is that the actual application exit is considered as an implementation detail and not as a separate component. The reason behind this decision is based on the observation that the generic solution is surprisingly often used.

\begin{verbatim}
trait ExitHandlerImpl extends HandlerComponent {
  override protected val handlerRepo : HandlerRepo = new ExitHandlerRepoImpl  // (X)

  private[this] class ExitHandlerRepoImpl extends HandlerRepo {  // (1)
    override def handle(b: Boolean, s: String) {  // (2)
      if (!b) {
        sys.error("CONTRACT VIOLATION: " + s)
        sys.error("Terminating application now!")
        System.exit(1)  // (3) replace it with a controlled shutdown
      }
    }
  }

Listing 6.7: f-DBC Exit handler implementation
\end{verbatim}
In case a project requires a considerably more sophisticated shut-down sequence, this can still be implemented as a custom component. The default exit handler only exists to emulate the default behaviour of countless other DBC implementations, including the one in Eiffel.

Taking full advantage of f-DBC’s profiles, allows using different handler for different situations. Taking a production profile as example, a system shut-down is not an option. Instead, commonly used solutions are email alerts or SNMP\textsuperscript{13} notifications. Taking email alert as one solution, f-DBC already provides by default an email alert handler, as shown in listing 6.8. This implementation imports a util\textsuperscript{(1)} based on apache commons\textsuperscript{14} that defines all the details for sending emails. However, the exact details of the email util are omitted\textsuperscript{15} in this case for keeping the focus on the notification handling. There are only three noticeable details in this implementation. First, email alerts are only send in case of a contract violation\textsuperscript{(2)}. Second, the syntax send a new Mail\textsuperscript{(3)} is defined in the email util. Third, the actual error message is provided in the message body.

```
trait EmailAlertHandlerImpl extends HandlerComponent {
  import util.mail. // (1)
  override protected val handlerRepo : HandlerRepo = new AlertHandlerRepoImpl

  private[this] class AlertHandlerRepoImpl extends HandlerRepo {
    override def handle( b: Boolean, s: String ) {
      if (!b) { // (2)
        send a new Mail( // (3)
          from = ("system@mycompany.com", "System Alert"),
          to = "admin@mycompany.com",
          subject = "ERROR: Contract violation",
          message = "Dear Admin, the check " + s + " has failed " // (4)
        )
      }
    }
  }
}
```

Listing 6.8: f-DBC email handler implementation

\textsuperscript{13}Simple Network Management Protocol (SNMP) is a widely used protocol for monitoring the health of network equipment: www.net-snmp.org/
\textsuperscript{14}http://commons.apache.org/proper/commons-email/
\textsuperscript{15}For the interested reader, all source code is provided through the companion website for this thesis at: http://marvin-hansen.github.io/Msc/
In case a different email system is required, the nested private class can be replaced with a custom implementation for any other kind of email system. Apart from the exit handler and the email handler, f-DBC provides two more handlers, a console handler that prints all passed and failed checks to the system console and a logger handler that writes all passed and failed checks into a log file using different log levels, depending on the case. The implementation details of the console handler are skipped as it is very similar to the logger handler and essentially only uses `println` instead of `loggerRepo` to print all messages to the system console.

The logger handler implementation, as shown in listing 6.9 has four noticeable details. First, the self type annotation\(^{16}\) adds the context of `LoggerComponent` to the scope of `this`. Through that, the abstract type member `loggerRepo` becomes accessible even though no specific logger component is given yet. Instead, `loggerRepo` only refers to type `LoggerRepo` (notice capital L) which is the protected nested trait that defines the actual interface of the logger component.

Second, this component has two internal implementations: One that logs all failed and passed checks and another one that only logs failed checks. The decision which one is used is defined by overriding the abstract type `handlerRepo` (2) with the chosen implementation.

Third, the first implementation that logs failed and passed checks uses different log levels. In case a contract violation is detected, the `error` log level is used to highlight the issue in the log file. If there is no contract violation, the check is logged using `info` log level. Logging both events is useful for testing and automated log analysis. In practice, log analyser tools\(^{17}\) that are often deployed to scan log files periodically for pattern such as error log level in order to trigger required actions. Also, during debugging it is important to know if a particular check has passed or not.

Fourth, the second implementation (4) only logs contract violations in order to keep log files small which is best practice for operational systems intended to run permanently.

\(^{16}\)see section 4.4.6 for details of self-types and section 4.8 for the cake pattern

\(^{17}\)For examples: [http://awstats.sourceforge.net/docs/awstats_compare.html](http://awstats.sourceforge.net/docs/awstats_compare.html)
Warne: This implementation requires a logger component for logging */

trait LoggedHandlerImpl extends HandlerComponent {
  // (1)
  this: LoggerComponent =>
  // (2)
  override protected val handlerRepo: HandlerRepo = new AllLogHandlerRepoImpl
  // (3)
  private[this] class AllLogHandlerRepoImpl extends HandlerRepo {
    override def handle(b: Boolean, s: String) {
      if (!b) { loggerRepo.error("CONTRACT VIOLATION: " + s) }
      else { loggerRepo.info("CHECK PASSED: " + s) }
    }
  }
}

Listing 6.9: f-DBC logger handler implementation

It is important to mention that only the logger handler implementation depends on a logger component but the handler component interface does not have any defined dependencies. In practice, this means using the handler component does not enforce any dependencies unless the logger handler is chosen as implementation. If the logger handler is composed without a logger implementation, a compiler warning is issued.
Logger component

The logger component follows the logging API\textsuperscript{18} specification from the Simple Logging Facade for Java (SLF4J). SLF4J serves as a facade for various logging frameworks allowing the end user to plug in the desired logging framework as required. There is one native implementation (logback) and four officially supported external libraries including the popular log4j library. Unlike many other logging frameworks, SLF4J supports the integration of loggers that are not compliant with the specified facade such as such as apache commons logging\textsuperscript{19}.

Making f-DBC compliant with the SLF4J API specification allows the choice of at least five different logging frameworks through one single implementation of the logging interface. The logger component interface is shown in listing 6.10. The interface defines the five standard log levels as specified by SLF4J. The error log level has an additional signature for including exception stack-traces in log messages.

```
/** Interface */
trait LoggerComponent {
  protected val loggerRepo: LoggerRepo
  protected trait LoggerRepo {
    def error(msg: => String, e: Throwable)
    def error(msg: => String)
    def info(msg: => String)
    def debug(msg: => String)
    def trace(msg: => String)
    def warn(msg: => String)
  }
}
```

Listing 6.10: f-DBC Logger interface

The *LoggerComponent* has two implementations. The first one is using SLF4J with the logback framework attached to it. The second one is a generic console logger which only prints all logs to the system console. Even though, logback, like any other SLF4J implementation supports console output, the console logger does not have any dependencies and does not require any configuration which makes it ideal for a fast development setup.

\textsuperscript{18}http://www.slf4j.org/apidocs/
\textsuperscript{19}http://commons.apache.org/proper/commons-logging/
The SLF4J implementation on the other hand allows flexible configuration of the chosen framework including connecting to a logging server thus makes it ideal for projects with more demanding requirements. The SLF4J implementation is shown in listing 6.11. A few details need closer attention.

First, the abstract type member `loggerRepo` is overridden with a specific instance of the logback implementation. This allows swapping the used logger library as well as managing centrally logger configurations.

Second, even though logback does not require any configuration for console logging, it allows fairly sophisticated configuration for different scenarios. The link \( (2) \) to the documentation provides all required details how to customize the default settings. In order keep the configuration isolated from other frameworks, a separate private nested class for the logback logger is the best choice. This is because other frameworks such as `log4j` have a different configuration approach thus the `SLF4JLoggerImpl` trait allows to manage all configurations in one central place. Furthermore, several nested private classes can be used as configuration for one logging framework.

Third, by importing the SLF4J `LoggerFactory`, the actual dependency is on the SLF4J facade and not on logback. This is because of the `plug-in` mechanism SLF4J uses. Changing the logger framework is means replacing a Jar file to one of the five supported logging framework. For instance, if high throughput and minimal latency is of high priority, the `log4j`\(^{20}\) which uses asynchronous logging based on the LMAX disruptor\(^{21}\) can be used as SLF4J backend.

The choice for logback as default logger is based on two specific details. On the one hand, there is no need for initial configuration thus logback works already out of the box. Also, the entire configuration for natively supported SLF4J logging frameworks such as logback is located either in a groovy or XML file within the class-path accessible to the application\(^{22}\). This means, the logback implementation is entirely free of any configuration details. On the other hand, logback is known to work well within any application container such as Tomcat thus the logback logger does not limit any deployment scenarios.

\(^{20}\)http://logging.apache.org/log4j/2.x/

\(^{21}\)For details about the disruptor framework, see section 4.11.7

\(^{22}\)http://logback.qos.ch/manual/configuration.html
trait SLF4JLoggerImpl extends LoggerComponent {
  // (1) chose private implementation here.
  override protected val loggerRepo: LoggerRepo = new LogbackLoggerRepoImpl

  /** (2) Details how to configure Logback: */
  * http://logback.qos.ch/manual/configuration.html */ */
  private[this] class LogbackLoggerRepoImpl extends LoggerRepo {
    // (3)
    import org.slf4j.LoggerFactory

    private[this] val log = { LoggerFactory.getLogger(getClass)} //(4)
    override def error(msg: => String, e: Throwable) {
      if (log.isEnabledFor(Error)) log.error(msg, e)
    }

    override def error(msg: => String) {
      if (log.isErrorEnabled) log.error(msg)
    }

    override def info(msg: => String) {
      if (log.isInfoEnabled) log.info(msg)
    }

    override def debug(msg: => String) {
      if (log.isDebugEnabled) log.debug(msg)
    }

    override def trace(msg: => String) {
      if (log.isTraceEnabled) log.trace(msg)
    }

    override def warn(msg: => String) {
      if (log.isWarnEnabled) log.warn(msg)
    }
  }
}

Listing 6.11: f-DBC SLF4J logger implementation

Fourth, making the instance of the logger (4) private[this] limits the scope not only to the actual instance of the component but also allows inline optimization during compilation.

Finally, the SLF4J facade requires a conditional check "if (log.isWarnEnabled)"(5) prior to logging which means, logging for each log level can be switched on and off using boolean flags.

Bringing all three core components, checker, handler and logger together is done on the composition layer, as described in the next section.
Chapter 6. f-DBC

### 6.6.2 Composition

The composition layer is the foundation for f-DBC profiles. There is one CompositionComponent with multiple implementations, each represent a different compositions. By default, f-DBC comes with three different compositions.

1. Checker handler composition
2. Logged checked handler composition
3. Mock composition

The important difference between a composition and a profile is that a composition only defines what to do but a profile specifies how to do the task defined in a composition. Also, several profiles can re-define a composition but each composition should be unique. Method naming is also part of a composition which means two different compositions can, for instance, express two different styles of contract writing whereas each style can be used in several profiles for different parts of a project.

By default, the composition interface, as shown in listing 6.12, defines only four methods. However, the interface in listing 6.12 is a compact representation; the complete interface has several overloaded signatures for each method to allow also arrays, list and sequences of boolean expressions. The only difference between all those signatures is the actual type for the boolean expression thus the redundant signatures have been excluded from listing 6.12 for brevity.

There are four methods defined in the composition interface. The first one, precondition, takes a description of the actual check and any boolean expression to check. Second, a postcondition that also takes a description and a boolean expression. Third, an invariant check, that takes two objects of any kind as parameter. Fourth, a conditional check that has an additional pre-condition that must be true before the actual check gets evaluated.

---

23Full source code at: [http://marvin-hansen.github.io/Msc/](http://marvin-hansen.github.io/Msc/)
Chapter 6. f-DBC

The default interface follows the naming convention introduced by Eiffel (see section 3.2). However, in a different context, this interface can be replaced with another one, for instance, following the naming and methods of ADA (see section 3.5).

```scala
/** Composition interface, method overloading is omitted */
trait CompositionComponent {
  protected val compositionRepo: Repo
  protected trait Repo {
    // (1)
    def precondition(@NotNull descr: String, @NotNull f: => Boolean)
    // (2)
    def postcondition(@NotNull descr: String, @NotNull f: => Boolean)
    // (3)
    def invariant(a: Any, b: Any) : Unit
    // (4)
    def conditionalCheck(@NotNull cond: Boolean,
                          @NotNull descr: String,
                          @NotNull b: => Boolean)
  }
}
```

**Listing 6.12**: f-DBC Composition interface

By default, f-DBC comes with three implementations. The first one is the `Checker-HandlerComposition` that composes a contract checker and contract violation handler together. The second implementation is the `LoggedCheckerHandlerComposition` that adds an additional logger so that each step is logged. The third one is a `Mock-Composition` that does exactly nothing by redirecting any method calls to empty implementations.

**Checker handler composition**

Beginning with the `CheckerHandlerComposition` (listing 6.13), there are several details that require close attention. First the self-type annotation\(^{24}\) (1) uses mixin-composition\(^{25}\) to redefine the scope of this to the checker and handler component. Thus, all abstract type members of both components are available within the scope of the `CheckerHandlerComposition`. Second, the nested private class `CheckerHandlerRepoImpl` (2) implementing the abstract type member definition `Repo` is a compact version of the real implementation.

---

\(^{24}\)For details see section 4.4.6

\(^{25}\)For details see section 4.7
Chapter 6. f-DBC

Just like the composition interface, all overloaded methods are omitted\(^\text{26}\) to keep the listing short.

Third, pre and postcondition (3) are implemented as a nested call of the checker function within the handle function. Resolving the call from the inside out, the innermost expression takes any boolean function and a description and passes it to the \textit{eval} function defined in the abstract type member definition \textit{checkerRepo} which then dispatches the \textit{eval} call to the used checker implementation. The eval call return either true of false which then gets passed to the \textit{handle} function defined in the \textit{handlerRepo} which dispatches the call to the used handler implementation. Depending on the actual handler implementation, further actions such as logging, email alert or application termination will be taken.

\begin{lstlisting}[language=scala]
trait CheckerHandlerComposition extends CompositionComponent {
  // (1)
  this: CheckerComponent with HandlerComponent =>
  override protected val compositionRepo: Repo = new CheckerHandlerRepoImpl
  // (2)
  private[this] class CheckerHandlerRepoImpl extends Repo {
    // (3)
    override def precondition(dscr: String, f: => Boolean) {
      handlerRepo.handle(checkerRepo.eval(f), dscr)
    }
    override def postcondition(dscr: String, f: => Boolean) {
      handlerRepo.handle(checkerRepo.eval(f), dscr)
    }
    // (4)
    override def invariant(a: Any, b: Any) = {
      handlerRepo.handle(checkerRepo.equal(a, b),
      "Handle invariant check: " + a + " must be: " + b)
    }
    // (5)
    override def conditionalCheck(cond: Boolean, dscr: String, b: => Boolean) {
      if (cond){handlerRepo.handle(checkerRepo.eval(b), dscr)}
    }
  }
}
\end{lstlisting}

\textbf{Listing 6.13:} f-DBC checked handler composition

Fourth, invariant is implemented by using the \textit{equal} function of the checker component. Consequentially, invariant only checks if a is equal to b. It is the developers responsibility to ensure that a and b are the correct values used expressing an in-

\(^{26}\)Full source code at: \url{http://marvin-hansen.github.io/Msc/}

201
variant check. In practice, this means an additional field \( \_OLD \) needs to be defined representing the initial value of the field to check. The actual invariant check compares then \( \_OLD \) to the actual field and if the values are unequal, the check fails.

Fifth, a conditional check is only triggered if the pre-condition is true. This means, reversing the effect requires negating the pre-condition inside the method call. In this implementation, a failed pre-condition will be ignored and passes through the check without any notice. For a more verbose checking, the logged implementation is the right

**Logged composition**

The \textit{LoggedCheckerHandlerComposition} resolves the issue if the exact contract passing or failure is required to know, for instance during debugging. Naturally, this makes the generated output more verbose but this usually helps to identify oblivious contract failures. An oblivious contract is one that never gets evaluated because of incorrectly defined boolean expressions. However, if a developer assumes this contract guards the underlying assertions but because the contract is never checked, the assertion is actually unprotected thus not verifiable. For instance, an accidentally negated pre-condition leads to the problem that the actual contract will never be evaluated thus the contract "slips through the net". Debugging those oblivious contracts is usually difficult because the trigger for checking the contract is not explicitly reporting whether a pre-condition has passed or not.

The primary difference that separates the \textit{LoggedCheckerHandlerComposition} from other implementations (and many existing DBC frameworks) is that it reports passed, failed and oblivious checks. Passed checks are logged in order to support a developer to decide if the right contracts have passed. Failed checks are reported so that a developer can explicitly verify if the right contracts have failed or not. Oblivious checks are covered by explicitly reporting each and every passed or failed pre-conditions thus a developer can instantly go to the source of a failed pre-condition and correct a potential incorrectly negated pre-condition. Without the explicit log of pre-conditions, such a task most likely ends up in a time consuming step by step debugging. From this perspective, switching to the logged composition is useful, whenever complications occur.
The implementation of precondition is shown in listing 6.14. The implementation of postcondition follows exactly the same technique thus postcondition is skipped here. Instead, the implementation details of precondition referring to both, pre and post- condition. For pre (1) and postcondition, the exact check gets logged first followed by the provided description of the actual check. Next, the check expression gets evaluated and if the result is true, a pass will be logged before handling the result. This is required because the chosen handler decides if or how to handle a passed check. If the result is false, a fail will be logged and the contract violation will be handled. Again, the actual action depends on the chosen handler.

```
trait LoggedCheckerHandlerComposition extends CompositionComponent {
  this: CheckerComponent with HandlerComponent with LoggerComponent =>
  override protected val compositionRepo: Repo = new LoggedCheckerHandleRepoImpl

  private[this] class LoggedCheckerHandleRepoImpl extends Repo {
    // (1)
    override def precondition(descr: String, f: => Boolean) {
      loggerRepo.info("Checking precondition: " + f.toString())
      loggerRepo.info("Details of precondition: " + descr)
      if (checkerRepo.eval(f)) {
        handlerRepo.handle(checkerRepo.eval(f), descr)
      } else {
        loggerRepo.error("FAILED: HANDLING CONTRACT VIOLATION")
        handlerRepo.handle(checkerRepo.eval(f), descr)
      }
    }

    // (2)
    override def precondition(descr: String, b: List[Boolean]) {preCheck(descr, b)}
    override def precondition(descr: String, b: Array[Boolean]) {preCheck(descr, b.toList)}
    override def precondition(descr: String, b: Boolean*) {preCheck(descr, b.toList)}
  }
```

Listing 6.14: f-DBC logged precondition

The first signature (1) handles check functions of any kind. However, in order to im- plement signature overloading efficiently, a preCheck method is defined in order to re- fer all remaining signatures (2) to the same check routine. Through that, lists, arrays and sequences of boolean expressions are supported as well which gives a developer choice how to define a contract. It is important to note that the preCheck method is declared private[this] which means the Scala compiler will inline the method which removes the extra method call.
Chapter 6. f-DBC

The \texttt{preCheck} method almost duplicates the first implementation (1) but instead of calling \texttt{eval} for function evaluation, it called \texttt{check} for checking boolean expressions. Through this private help method, the public API of the compose interface can expose any type of boolean expression while still preserving efficient implementation of all the different signatures. Similar signature overriding exists for postcondition and conditional checks but they all follow this implementation technique to minimize code duplication.

Invariant has also a fully logged implementation, as shown in listing 6.15. Just like pre and postcondition, the actual invariant check gets logged by explicitly stating the expected value. If the check passes, a pass will be logged before handling the result. Likewise, a failed check informs that the value has changed before handling the failed result.

```java
override def invariant(a: Any, b: Any) {
    loggerRepo.info("Checking invariance: " + a + " must be: " + b)
    if (checkerRepo.equal(a, b)) {
        loggerRepo.info("PASSED: " + a + " is invariant")
        handlerRepo.handle(checkerRepo.equal(a, b), "PASSED INARIANT")
    } else {
        loggerRepo.error("FAILED: " + a + " has changed, handling INVENTORY VIOLATION")
        handlerRepo.handle(checkerRepo.equal(a, b), "INVENTORY VIOLATION")
    }
}
```

\textbf{Listing 6.15:} f-DBC: logged invariant check

The logged conditional check (listing 6.16) implements the same approach by first checking the precondition. If the pre-condition passes, both the condition and the success are logged before checking the actual contract. If the pre-condition fails, the failure and the condition are logged before informing that no further check will be performed. Similar, if the actual contract passes the check, the success will be logged before handling the result. Finally, if the contract check fails, the failure and contract violation will be logged as well. It is important to notice the different log applied. For passed checks, the log level is info but for all failed checks, the log level is error. This supports automated log file analysis as well as automated error processing, depending on the actual work-flow of a project.
overridded def conditionalCheck(cond: Boolean, descr: String, b: => Boolean) {
  loggerRepo.info("Checking conditional check: " + descr)
  loggerRepo.info("Checking required pre-condition: " + cond)
  if (cond) {
    loggerRepo.info("PASSED: pre-condition " + cond + " is true")
    loggerRepo.info("Checking conditional check: " + b.toString)
    loggerRepo.info("Details of conditional check: " + descr)
    if (checkerRepo.eval(b)){
      loggerRepo.info("CHECK PASSED: Handling result now")
      handlerRepo.handle(checkerRepo.eval(b), descr)
    } else{
      loggerRepo.error("CHECK FAILED: HANDLING CONTRACT VIOLATION")
    }
  } else{
    loggerRepo.error("FAILED: pre-condition " + cond + " is FALSE")
    loggerRepo.error("ABORT CHECK: No further evaluation will be performed")
  }
}

Listing 6.16: f-DBC: logged invariant check

Mock composition

The last implementation of the composition interface is the `MockComposition` which redirects any method call to an empty method so that no action will be performed. The underlying intention is to use profiles and aliases to temporary switching off checking all contracts either for selected modules or the entire project. A minimal implementation, without signature overriding, is shown in listing 6.17.

trait MockComposition extends CompositionComponent {
  override protected val compositionRepo: Repo = new MockRepoImpl
  private[this] class MockRepoImpl extends Repo {
    override def precondition(descr: String, f: => Boolean) {}
    override def postcondition(descr: String, f: => Boolean) {}
    ....
  }

Listing 6.17: f-DBC: Mock composition

The composition needs to be combined with mock components in order to define a profile that can be used in a project. The next section provides more details about how to define and use profiles.
6.6.3 Profiles

Profiles separate f-DBC from all other existing DBC solutions available. Essentially, a profile is the combination of one composition and a set of specific implementations that specify how the composed task is to accomplish. By default, f-DBC already comes with five profiles for development, debugging, testing, mock testing and production.

Default profiles

All default profiles are using a sequential checker in order to preserve the exact order of expression checking. Parallel checker should only be used if the number of expressions is large and checks can be performed in any order. Each default profile reflects a different configuration depending on the actual requirements. Table 6.3 summarizes the different configurations and used components for each profile.

The default development profile uses an exit handler which means any contract violation simply terminates the application. This is useful to detect faults as early as possible. The default debugging profile takes a different approach and keeps the application running while only logging contract violations to the system console. This means, the debugging profile does not interfere with any other debugger applied to a project but a developer knows, whenever a message appears to the system console, a contract violation happened.

<table>
<thead>
<tr>
<th>Profile</th>
<th>Composition</th>
<th>Checker</th>
<th>Handler</th>
<th>Logger</th>
</tr>
</thead>
<tbody>
<tr>
<td>Development</td>
<td>CheckerHandler</td>
<td>SequentialChecker</td>
<td>ExitHandler</td>
<td>N/A</td>
</tr>
<tr>
<td>Debugging</td>
<td>CheckerHandler</td>
<td>SequentialChecker</td>
<td>ErrorLoggedHandler</td>
<td>ConsoleLogger</td>
</tr>
<tr>
<td>Testing</td>
<td>LoggedCheckerHandler</td>
<td>SequentialChecker</td>
<td>AllLoggedHandler</td>
<td>SLF4JLogger</td>
</tr>
<tr>
<td>Production</td>
<td>LoggedCheckerHandler</td>
<td>SequentialChecker</td>
<td>EmailAlertHandler</td>
<td>SLF4JLogger</td>
</tr>
<tr>
<td>Mock</td>
<td>MockComposition</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Table 6.3: f-DBC: Default profiles

The default testing profile uses a logged composition which means each and every check gets logged in detail. Combined with the SLF4 logger, any of the four supported log frameworks can be used. Note, the SLF4 component already comes pre-configured with a logback implementation that only needs to be customized.
Also, the `AllLoggedHandler` logs passed and failed checks without taking further action. This particular behaviour is important in order to prevent interference with automated testing or continuous integration. For reducing the verbosity of logging, the `AllLoggedHandler` can be replaced with the `ErrorLoggedHandlerImpl` which only logs contract violations.

The default production profile uses the SLF4 logger as well but in combination with an email alert handler. It is important to note that the email handler only report failed checks and does not take any further action which means the application keeps running despite a contract violation. This is motivated by keeping a production system running while preserving the ability to spot contract violations which may only occur in very specific situations. Also, this is in line with the common practice to patch production systems only on schedule in a planned take-over process.

**Custom profiles**

Default profiles in f-DBC can be customized at any level which means custom implementation of existing components can be used, new custom components can be added or even entirely new compositions can be created as required. The default profile only exist to cover the most commonly scenarios in a software project. The provided default implementations should be sufficient to create better matching custom profiles in case a different contract handling is required. By definition, any number of profiles can be created in f-DBC. Creating a custom profile in f-DBC requires three steps.

First, an composed interface needs to be created in order to match the requirements defined in a composed interface. This composed interface trait is called a `Registry` because it registers several components to one composition component. Taking the development profile as example, the corresponding registry is shown in listing 6.18. Essentially, the registry matches the `CompositionComponent` with all component interfaces required for the chosen implementation. In the example, this means the checker and the handler component are required.

Second, a composed implementation needs to be created by mixing specific implementations to one specific composition. This composed implementation trait is called a `RegistryImpl` because it composes specific implementation to the composed interface.
Chapter 6. f-DBC

1 /* composed interface */
2 protected trait DevRegistry extends CompositionComponent
3 with CheckerComponent with HandlerComponent

Listing 6.18: f-DBC Dev. profile registry

It is important to note, that each Registry can have several RegistryImpl in order to swap composed implementations as required. Listing 6.19 shows the Registry-Impl for the development profile which matches the chosen implementations to the selected composition. As the example shows, the exit handler is used together with the sequential checker in order to specify implementations for the CheckerHandler-Composition.

1 /* composed implementation */
2 protected trait DevRegistryImpl extends DevRegistry
3 with CheckerHandlerComposition with SequentialCheckerImpl
4 with ExitHandlerImpl

Listing 6.19: f-DBC Dev. profile implementation

Third, a mixin trait and a corresponding companion object needs to be created. The mixin trait mixes the Registry and one RegistryImpl together in on single trait. The RegistryImpl can be replaced at any time as long as the same Registry is used. The companion object inherits from the mixin trait in order to provide static access to the mixin trait. This is required for to preserve full compatibility with Java by using import to access an instance of the mixin trait. Following the example, listing 6.20 shows the mixin trait that mixes the DevRegistry together with the DevRegistryImpl and defines a corresponding companion object. At this point, any implementation of the DevRegistry can be used.

1 object DevelopmentProfile extends DevelopmentProfile
2 /* composite trait, matches interface to implementation by name */
3 trait DevelopmentProfile extends DevRegistry with DevRegistryImpl

Listing 6.20: f-DBC Dev. profile and companion object
Profiles are already useful and a step forward in order to make contract handling more flexible to use but sometimes real requirements demand more flexibility. For instance, once a project has left the testing stage, some parts may get deployed but other parts require more testing. Following testing, a different production profile may become required because the production system may have changed in the meantime. In order to cope with progressing projects and changing requirements, aliases are used on top of profiles.

6.6.4 Aliasing

An alias is a simple yet powerful way to swap profiles on demand whenever changes are required. Essentially, the core idea is using profiles only through an alias for each part of a project. Through that alias, staging is accomplished by swapping the profile connected to the alias. Previously, companion objects where defined per profile. Using aliases, however, requires the removal of all profile companion objects and that each profile is protected. After those changes are applied, the four profiles are defines as shown in listing 6.21. The removal of companion objects prevents accidental direct usage of profiles and making profiles protected limits the scope so that only aliases can access them.

```scala
1 protected trait DevelopmentProfile extends DevRegistry with DevRegistryImpl
2 protected trait TestingProfile extends TestRegistry with TestRegistryImpl
3 protected trait ProductionProfile extends ProductionRegistry with ProductionRegistryImpl
4 protected trait MockProfile extends MockRegistry with MockRegistryImpl
```

Listing 6.21: f-DBC protected profiles

The MVC pattern\(^{27}\) is commonly used to separate business logic from a graphical user interface (GUI) and the data store. Each of those three architectural layers has different requirements and different stages in development. Listing 6.22 shows four aliases and corresponding companion objects for each of the different layers in the MVC pattern. The first one is a generic alias that covers “everything else” which refers to components and modules without any specific requirements. Nevertheless, even the generic alias supports staging which means the development profile can be swapped for testing or production at any time. The second alias is a specific one for

---

\(^{27}\)Model View Controller (p. 342 in [91])
Graphical User Interfaces (GUI) which is linked to a mock profile in order to switch contracts off during prototyping. Once the prototyping phase is over, a specific gui-logger profile may replace the mock profile. The third alias is specifically for the business logic which is already in testing thus is linked to the testing profile. Once the testing is over, the testing profile needs to be replaced with a production profile. The last alias is especially defined for the used persistence (model in MVC terms) which is linked to the development profile.

```java
// (1) Generic alias
object fDBC extends fDBC

// (2) GUI specific alias
object GuiDBC extends GuiDBC

// (3) alias for business logic
object ModelDBC extends ModelDBC

// (4) Specific alias for data layer
object PersistenceDBC extends PersistenceDBC

Listing 6.22: f-DBC aliases
```

Using those aliases allows staging each module separately in a project. Furthermore, swapping implementations or profiles does not affect other modules. The later is important because it means specific profiles can be applied only to specific parts of a project monitoring particularly error prone components closely exposing problems as early as possible.

### 6.6.5 Extending f-DBC

Changing a composition does not affect defined contracts as longs as the composition interface is preserved. Extending the composition interface is possible as long as all existing signatures remain unchanged for compatibility with all existing compositions and profiles. Adding new method ore further refinements of existing will be accessible in the moment a new composition is replacing the existing composition of the `dbc` alias.
Chapter 6. f-DBC

This simplifies code maintenance significantly and allows applying gradually new releases of a new composition because existing code remains unchanged whereas newly written code can already use the new functionality provided by a new composition. Swapping a specific implementation of a profile, however, affects all code equally and shall be done carefully. Alternatively, a new profile with a new alias can be created as well in order to reduce the impact of significant changes in a profile. Through that, "legacy support" is already built into f-DBC by allowing running several versions of a profile using different aliases. Also, migration support is also provided by simply swapping a profile linked to a profile. In the used account example, a new alias `accountDBC` may be used instead of the previously used `fDBC` alias. The only required change would be the replacement of the import statement or the mixed in trait respectively. This is possible because all methods are accessed through the abstract type alias `dbc` which is defined in the profile interface to be of type `Repo` as defined in the composition interface. This means, no matter which alias, profile or composition is used; the type and methods provided through the `dbc` type alias remain the same under all circumstances.

One remaining question is: How f-DBC would affect run-time performance of an application? The next section (7) investigates a case study for estimating the performance penalty involved by applying contracts to Scala.
Chapter 7

CBR Case study

7.1 Overview

In order to estimate the overall impact of f-DBC on run-time performance, I did a concurrency case study on a case based reasoning on travel data. In cased based reasoning (CBR), ”a new problem is solved by finding a similar past case, and reusing it in the new problem situation” [125] I have chosen CBR as foundation for my concurrency case study, because a case reasoning algorithm, correctly implemented, exploits task and data parallelism. This is a unique scenario because in practice, either task or data parallelism can be applied to a project, but having a scenario that applies both is rather an exception. Often, dependencies between data prevent the application of data parallelism. However, the benefit of exploring task and data parallelism on the same case study is a quantification of the exact benefit for each kind of parallelism on a specific scenario.

\footnote{Section 4.11.1 and provides more details about task and data parallelism.}


7.2 Data and methodology

At the heart of any CBR system there is a similarity function that quantifies the similarity of cases [126]. In CBR, there are three categories of similarity models: absolute, relative and metric similarity [127]. I’ve chose a metric model that calculates a global similarity score, based on the k nearest neighbour (K-NN) algorithm. The applied formula is:

\[
\text{Similarity}(T, S) = \frac{\sum_{i=1}^{n} f(T_i, S_i) \times W_i}{\sum_{i=1}^{n} W_i}
\]

In the formula, T is the target case, S is the source case, n is the number of attributes in each case and index i refers to an individual attribute from 1 to n. Function f is a local similarity function for attribute i in cases T and S. Each local similarity score is multiplied by its corresponding weight W, reflecting its relevance. The global similarity score is defined as the sum of all weighted local similarities, divided by the sum of all local weights. The division by the sum of all local weights is used for normalizing the global score.

The data-set is a collection of 1024 cases of arbitrary travel bookings. Each data record has twelve attributes and each attribute has corresponding local similarity function used calculating the distance between two values of the same attribute. Overall, five different concurrency implementations have been measured in addition to a non-concurrent implementation used as reference. Measurement has been made for the following concurrency approaches:

1. Parallel collection
2. Futures and for-comprehensions
3. Futures nested in parallel collections
4. Actors

Furthermore, the impact of SIMD data parallelism on task concurrency has been evaluated as well.
Chapter 7. CBR Case study

The disruptor implementation is only available as a SIMD version, because of the ring-buffer data-structure. Even though the disruptor is an event based approach similar to actors, the underlying ring-buffer uses a fixed-sized pre-allocated immortal memory structure that is significantly simpler to program with SIMD data parallelism.

The contract used for measuring the performance impact of f-CBC is written in a separate trait mixed into each implementation to ensure the same contract is used for performance evaluation. Listing 7.1 shows an excerpt of the used contract. The full source listing is in appendix C. Three details need to be mentioned. First, the contract uses an alias (1) to the fDBC alias, which is linked to the development profile. This means, all contracts are checked, and in case of a contract violation, an exception is thrown before terminating the application. This profile was chosen on purpose, since it is very similar to the production profile, which only differs in how exactly contract violations are handled. Therefore, it reflects any potential run-time overhead of used checks only.

```
1 trait alias extends fDBC // (1)
2 trait ReasonerSpec extends CaseReasonerI with ConfigManager with alias {
3   val _DBC_CHECKS = getDbcFlag // (2)
4   abstract override def getMostSimilarCase(refCase: Case,
5       cm: CaseManager, weights: FeatureWeights): Case = {
6     if (_DBC_CHECKS) {
7       dbc.precondition("Reference case must not be null", refCase != null)
8       dbc.precondition("Case manager must not be null", cm != null)
9     } // (3a)
10     super.getMostSimilarCase(refCase, cm, weights)
11   }
12 ...}
13 } // composition example:
14 class CaseReasoner_Futures extends CaseReasoner_FuturesImpl with ReasonerSpec
```

Listing 7.1: f-DBC contract

Second, contracting is switched on and off by parsing a command line parameter, which is assigned to a boolean flag (2) that indicates contract usage. Only if the flag is set to true, the contracts are checked before invoking the actual method (3a). Because the contract is inheriting from the same interface CaseReasonerI all concurrency traits are implementing, the `super` call (3a) gets dispatched from the interface to the correct implementation.
The composition example at the bottom of listing 7.1 shows how a particular concurrency implementation is mixed together with the contract in a new class. This composition is applied to all measured implementation.

A benchmark is used for executing a standard test case that compares each travel case to the remaining 1023 cases and selects the most similar one, which is performed for all 1024 cases. The benchmark application used for measuring performance is a custom framework I’ve written following the best practice from Boyer et al. [128]. For statistical accuracy, a pre-run is executing the task 80 times before measuring 10 runs of the task. The long pre-run is motivated by excluding JVM optimization, such as JIT compiling. Each of the 100 runs is measured independently to determine the minimum, maximum and mean execution time. Furthermore, variance and standard derivation are calculated in order to estimate the level of confidence in the results.

Execution time is measured in nanoseconds to circumvent an accuracy bug in System.currentTimeMillis(). Execution time is then converted to seconds by using a precise custom conversion to prevent loss of accuracy. All measurements are exported to an excel sheet for further evaluation.

7.3 Hypotheses

The first hypothesis is that the usage of f-DBC does affect the runtime performance of the case study. If the measurements of f-DBC versus no contracts differ by more than 0.1% from each other, then performance impact is considered.

The second hypothesis is that concurrency applied to the the CBR algorithm improves overall execution performance. An improvement is considered if the measurements of concurrency versus non-concurrency differs by more than 15% from each other. Furthermore, SIMD data parallelism is assumed to have an impact on performance. An impact is considered if the measurements of SIMD versus non-SIMD differ by more than 15% from each other.
Chapter 7. CBR Case study

7.4 Results

7.4.1 f-DBC performance impact on task parallelism

The investigation of the first hypothesis, the performance impact of f-DBC, has lead to the surprising result of no significant impact. The measurements are summarized in table 7.1. All measurements are below the threshold of 0.1% difference in execution times, which means, any potential difference is within the range of measurement tolerance.

<table>
<thead>
<tr>
<th>Task parallelism</th>
<th>Metric</th>
<th>f-DBC</th>
<th>No Contract</th>
<th>Perc. Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Min</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0%</td>
</tr>
<tr>
<td></td>
<td>Max</td>
<td>2.195860529</td>
<td>2.20086603</td>
<td>0.04832794 %</td>
</tr>
<tr>
<td></td>
<td>Avg</td>
<td>1.507086950</td>
<td>1.50917426</td>
<td>0.02274456 %</td>
</tr>
<tr>
<td></td>
<td>Var</td>
<td>0.028102039</td>
<td>0.02868863</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>Std</td>
<td>0.167636629</td>
<td>0.16937718</td>
<td>N/A</td>
</tr>
</tbody>
</table>

| Par. Col.        | Min    | 0.0       | 0.0         | 0.0%            |
|                  | Max    | 1.144725362 | 1.10737128 | 0.01267635%     |
|                  | Avg    | 0.735738122 | 0.738589491| 0.00543408%     |
|                  | Var    | 0.007708756 | 0.007635830| N/A             |
|                  | Std    | 0.087799521 | 0.087383239| N/A             |

| Futures          | Min    | 0.0       | 0.0         | 0.0%            |
|                  | Max    | 2.17800592 | 2.145650332| 0.03980351%     |
|                  | Avg    | 1.85507930 | 1.84328262 | 0.00065573%     |
|                  | Var    | 0.03557531 | 0.035007496| N/A             |
|                  | Std    | 0.18861418 | 0.187102903| N/A             |

| Fut. Par. Col.   | Min    | 0.0       | 0.0         | 0.0%            |
|                  | Max    | 0.964606888 | 0.980242521| 0.00945548%     |
|                  | Avg    | 0.855959608 | 0.860387002| 0.00736456%     |
|                  | Var    | 0.008788421 | 0.008946811| N/A             |
|                  | Std    | 0.093746579 | 0.094587586| N/A             |

| Actors           | Min    | 0.0       | 0.0         | 0.0%            |
|                  | Max    | 2.91069256 | 3.198054045| 0.09308552%     |
|                  | Avg    | 1.15257016 | 1.152421299| 0.01328246%     |
|                  | Var    | 0.08329552 | 0.120188205| N/A             |
|                  | Std    | 0.28860964 | 0.346681707| N/A             |

Table 7.1: f-DBC: Performance Evaluation
### 7.4.2 f-DBC performance impact on data parallelism

Similar to the previous result, f-DBC has no measurable impact on applied data parallelism. None of the measurements exceeds the threshold of 0.1% difference in execution time. One note, the measurements for Akka are missing because Akka crashed constantly with a timeout exception, which means an actor was not able to process one case comparison within the set time limit of three seconds. Previous measurements with a lower number of benchmark iterations confirm the general observation of f-DBC having only a negligible (under 0.01%) impact on Akka’s runtime performance.

<table>
<thead>
<tr>
<th>Data parallelism</th>
<th>Metric</th>
<th>Contract</th>
<th>No Contract</th>
<th>Perc. Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear (SIMD)</td>
<td>Min</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0%</td>
</tr>
<tr>
<td></td>
<td>Max</td>
<td>1.434538536</td>
<td>1.440037507</td>
<td>0.02065789 %</td>
</tr>
<tr>
<td></td>
<td>Avg</td>
<td>1.074992824</td>
<td>1.088953867</td>
<td>0.01170617 %</td>
</tr>
<tr>
<td></td>
<td>Var</td>
<td>0.013089366</td>
<td>0.013400409</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>Std</td>
<td>0.114408769</td>
<td>0.115760137</td>
<td>N/A</td>
</tr>
<tr>
<td>Par. Col. (SIMD)</td>
<td>Min</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0%</td>
</tr>
<tr>
<td></td>
<td>Max</td>
<td>0.657403842</td>
<td>0.664531036</td>
<td>0.00436865%</td>
</tr>
<tr>
<td></td>
<td>Avg</td>
<td>0.537809628</td>
<td>0.558720117</td>
<td>0.00300485%</td>
</tr>
<tr>
<td></td>
<td>Var</td>
<td>0.003541164</td>
<td>0.003407786</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>Std</td>
<td>0.059507683</td>
<td>0.058376250</td>
<td>N/A</td>
</tr>
<tr>
<td>Futures (SIMD)</td>
<td>Min</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0%</td>
</tr>
<tr>
<td></td>
<td>Max</td>
<td>1.422080322</td>
<td>1.622833687</td>
<td>0.02307799%</td>
</tr>
<tr>
<td></td>
<td>Avg</td>
<td>1.310120536</td>
<td>1.357142319</td>
<td>0.01778020%</td>
</tr>
<tr>
<td></td>
<td>Var</td>
<td>0.018424336</td>
<td>0.020322629</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>Std</td>
<td>0.135736275</td>
<td>0.142557459</td>
<td>N/A</td>
</tr>
<tr>
<td>Fut. Par. Col. (SIMD)</td>
<td>Min</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0%</td>
</tr>
<tr>
<td></td>
<td>Max</td>
<td>0.613356859</td>
<td>0.652423206</td>
<td>0.00400168%</td>
</tr>
<tr>
<td></td>
<td>Avg</td>
<td>0.555265787</td>
<td>0.580459278</td>
<td>0.00322309%</td>
</tr>
<tr>
<td></td>
<td>Var</td>
<td>0.003335135</td>
<td>0.003637061</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>Std</td>
<td>0.057750632</td>
<td>0.060308050</td>
<td>N/A</td>
</tr>
<tr>
<td>Disruptor (SIMD)</td>
<td>Min</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0%</td>
</tr>
<tr>
<td></td>
<td>Max</td>
<td>0.746350376</td>
<td>0.783539141</td>
<td>0.00584794%</td>
</tr>
<tr>
<td></td>
<td>Avg</td>
<td>0.491374875</td>
<td>0.493792775</td>
<td>0.00242725%</td>
</tr>
<tr>
<td></td>
<td>Var</td>
<td>0.003726139</td>
<td>0.003823723</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>Std</td>
<td>0.061042114</td>
<td>0.061836266</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Table 7.2: f-DBC: Performance Evaluation
7.4.3 Task and data parallelism performance impact

Verifying the second hypothesis has lead to a set of diverse results, as shown in table 7.3. Taking the base version (linear), without any concurrency applied, and calculating a similarity score for each of the 1024 cases takes 1.5 seconds on average. Actors require almost the same time, but have one magnitude higher variance and a one third higher maximum execution time. Similarly, Futures are in all measurements slower, compared to no concurrency applied. Parallel collection, on the other hand, have half the maximum, half the average execution time and a magnitude lower variance. The same applies for Futures nested in a parallel collection.

<table>
<thead>
<tr>
<th>Execution time (sec.) for task parallelism</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max.</td>
</tr>
<tr>
<td>------</td>
</tr>
<tr>
<td>Linear</td>
</tr>
<tr>
<td>Actors</td>
</tr>
<tr>
<td>Futures</td>
</tr>
<tr>
<td>Par. Col.</td>
</tr>
<tr>
<td>Fut. Par. Col.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Execution time (sec.) for SIMD data parallelism</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear (SIMD)</td>
</tr>
<tr>
<td>Actors* (SIMD)</td>
</tr>
<tr>
<td>Futures (SIMD)</td>
</tr>
<tr>
<td>Fut. Par. Col. (SIMD)</td>
</tr>
<tr>
<td>Par. Col. (SIMD)</td>
</tr>
<tr>
<td>Disruptor (SIMD)</td>
</tr>
</tbody>
</table>

Table 7.3: Concurrency performance Evaluation.

\(^a\)LoC includes comments and blank lines

Adding SIMD data parallelism, however, leads to noticeably better performance on the same quad core system for nearly any implementation with the exception of actors. For this comparison, the same data was used with the exception of actors, because no SIMD Actor data were available from the previous benchmark. Instead, data from a second benchmark run with fewer iterations were used for both, SIMD and non-SIMD actors. The general observation is, average execution time improves for all implementation with the exception of actors.
Chapter 7. CBR Case study

Figure 7.1 plots the data from table 7.3 to illustrate the actual performance gain for each concurrent implementation. There are three main observations. First, actors and Futures do not perform any better compared to the linear base version. Second, parallel collections and Futures nested in parallel collections perform noticeably better compared to the linear version.

![Comparison execution time](image)

**Figure 7.1:** Performance plot of all implementations

Third, SIMD improves performance further with the exception of actors. Putting those numbers into perspective, a complete run of the benchmark takes 150 seconds for the linear base version. Futures nested in parallel collections with SIMD take only 53 seconds, which is very close to the 49 seconds of the disruptor implementation. In other words, applying SIMD to a CBR system executes the same algorithm three times faster on the same hardware. The overall gain of SIMD is remarkably similar for all implementations except for actors. Figure 7.2 shows the actual performance gain or loss of the SIMD version over the non-SIMD in percent. For the linear base version, applying SIMD improves maximum execution time by 66 %, average execution time by 72 % and variance by 48% compared to the non-SIMD version.
The general observation is that the maximum execution time improves between 60 and 76% and the average execution time improve between 68 and 90% percent. Variance improves between 41 and 48%, as shown in figure 7.2.

Even though actors have a significant higher maximum execution time with SIMD compared to no-SIMD, the average execution time shows the highest degree of improvement. This observation appears to be a contraindication, but the same data plot shows a higher level of variance for SIMD, which means the difference between the mean and the maximum value is much higher for SIMD versus non-SIMD. More precisely, for actors variance has reduced by 254% through SIMD. My guess on this result is that SIMD does improve actor performance on average because SIMD minimizes cache misses on CPU level. However, if an actor processes messages ”too fast”, too many tasks are pushed to the underlying task execution pool leading to thread congestion reducing the maximum execution time significantly.

Another possible explanation could be that the overall inefficient actor model already has a high level of cache misses at CPU level and adding SIMD pushes even more data to the CPU, which increases the cache miss rate instead of reducing it.

Regardless of the underlying reasons, the improved mean execution time for SIMD actors is unlikely to be achievable in practice because of the high level of variance.
7.4.4 Scalability of task and data parallelism

One question that emerged during analysing performance in Scala is: How does each implementation scales with the number of cores in a system? For instance, would increasing the number of cores lead to an equal rise in performance? Also, how does variance change with the number of cores?

In order to answer these question about scalability, I compared the previous benchmark results from a quad core Intel I3 with an additional benchmark on a quad-socket quad-core AMD Opteron server\(^2\), having altogether 16 cores.

For this experiment, the level of parallelism changes from four to sixteen cores, which is a four time increase that, in theory, would lead to a four time increase in performance. In practice, however, a certain overhead for managing parallel tasks is required; therefore the theoretical maximum increase in performance cannot be reached. However, the question is, how much each of the evaluated concurrency frameworks can approximate the theoretical limit.

In this context, variance is very important, because a higher level of parallelism only leads to a reliable high level of performance if the overall execution time has a low level of variance. A high level of variance, on the other hand, would nullify most of the gained performance, because it reduces the average execution time down to a similar level of linear execution.

As the previous results clearly stated, SIMD performs better for all cases except actors. Therefore, I’ve only included SIMD versions of all versions except for the linear and actor implementation. I have disabled f-DBC in order to avoid confusion with real and arbitrary measurement tolerance. Also, a pre-experiment has confirmed the same low level (under 0.1\%) of performance difference between f-DBC versus contracts disabled; therefore disabling contracts entirely do not imply any loss of performance information.

Based on the experience that the actor implementation crashes if the benchmark exceeds 150 or more total iterations, I’ve changed the configuration. The benchmark was executed with 40 pre-runs and 60 measured iterations preventing the actor implementation from crashing.

\(^2\)This system was rented from Amazon Web Services (AWS).
Chapter 7. CBR Case study

Figure 7.3 summarizes the results presented as change in percentage of 16 cores over 4 cores. A few observations: First, the slight reduction in execution time for the linear base version is no error, but based on the fact that server CPU’s are 2012 Opterons and the second system uses a 2013 quad-core i3, which means, within a year, linear execution time has been improved. This difference was expected, but does not affect the general observation of scalability.

![Percentage difference in execution time 4 vs. 16 cores](image)

**Figure 7.3:** Performance SIMD: 4 vs. 16 cores

Second, the previous observation of Futures not performing any better compared to the linear version remains true on the 16 core system. Third, actors without SIMD scale noticeably better on a 16 core system compare to actors with SIMD. In particular, actors without SIMD improve variance by 794 % over the quad-core system, whereas the SIMD implementation only improves variance by 328 %. This difference is important, since better variance leads to better average performance. The disruptor implementation scales by improving average execution time twice while improving variance threefold. The previous observation of parallel collections and nested Futures performing on a similar level compared to the disruptor remains true on the 16 core system.
Table 7.4 summarizes the measurements of nested Futures and the disruptor on both, the quad and 16 core systems.

<table>
<thead>
<tr>
<th></th>
<th>4 Cores</th>
<th>16 Cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disruptor execution time (sec.)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Max.</td>
<td>0.732170184</td>
<td>0.384512515</td>
</tr>
<tr>
<td>Avg.</td>
<td>0.525593869</td>
<td>0.184436301</td>
</tr>
<tr>
<td>Var.</td>
<td>0.006260711</td>
<td>0.001461086</td>
</tr>
<tr>
<td>Nested Futures exec. time (sec.)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Max.</td>
<td>0.631045398</td>
<td>0.261964986</td>
</tr>
<tr>
<td>Avg.</td>
<td>0.582753049</td>
<td>0.193596420</td>
</tr>
<tr>
<td>Var.</td>
<td>0.005802114</td>
<td>0.000775422</td>
</tr>
</tbody>
</table>

(a) Disruptor: 4 vs. 16 Cores

(b) nested Futures: 4 vs. 16 Cores

Table 7.4: Comparison between disruptor and nested Futures

This observation is particularly important, because the disruptor framework is by any measure the fastest possible concurrency approach available for Java. Considering the fact that parallel collections and nested Futures only require one fifth of LoC of the disruptor, this means a very similar level of performance can be achieved with significantly lower development effort and time.

7.5 Analysis

The result provide three major insights. First, f-DBC does not affect performance of the CBR system. Second, SIMD data parallelism improves performance of all implementation with the exception of actors. Third, functional concurrency in Scala can perform as fast as the fastest concurrency in Java while only requiring one fifth of code lines.

The first result shows that the performance impact of f-DBC can hardly be quantified for any linear, concurrent or SIMD implementation. Furthermore, the marginally better measurements for f-DBC do not indicate a better performance, but refer to a non-explainable phenomenon of the JVM I have observed frequently during the experiments. Essentially, the difference reverses, if the two benchmarks are executed in reversed order. Even though a new JVM instance is used for each run of the benchmark, the second run always performs insignificantly better. As the JVM is widely believed to discard all profiling data at termination, I cannot explain this particular observation at this stage.
As this execution time variation is usually within the range 0.05% and 0.1%, the threshold of 0.1% difference in run-time performance was chosen on purpose for excluding this phenomenon. One particular implication of this result is the indirect evidence that using the cake pattern does not affect run-time performance. It appears to me, overriding abstract types and using mixing composition is well optimized by the Scala compiler. Essentially, this observation means that using good software design in form of component based dependency management through the cake pattern does not affect run-time performance.

The second observation showed that SIMD data parallelism improves average execution time of the CBR benchmark between 68 and 90% percent. This result does not surprise, since SIMD essentially reduces cache misses at CPU level within the same range. As CPU cache is accessed within one digit nano-seconds, loading more data from it leads consequently to better performance.

However, the significant performance reduction of the actor implementation does surprise me. Even though the actor implementation is by no means a stellar performer, overly poor impact of SIMD raises some questions about the efficiency of actors. Also, the issue of reproducible crashes and observed excessive memory usage (over 6GB) questions the efficiency of the actor approach even more.

This observation supports the thought of SIMD actors being more prone to CPU cache misses compared to non-SIMD. Excessive memory usage means also excessive CPU cache usage, which leads, if CPU cache is full, to more cache misses. There are three details in the benchmark setup supporting this thought. First, on the 16 core system, each Opteron CPU has 6 MB of L3 CPU cache, which explains the enormous improvement in variance for non-SIMD actors versus the quad core i3 CPU, which has only 3MB of L3 Cache. Second, adding SIMD increases the pressure on CPU cache noticeably.

A completely optimized SIMD data structure containing all data for all cases has approximately 2.8 MB, which occupies nearly half of the 6MB L3 cache in the 16 core system and nearly all of the L3 cache in the quad-core i3 system. This increases cache hits noticeably, which the measurements for SIMD actors confirm. Third, the remaining concurrency frameworks are not affected by this because they have an order of magnitudes lower memory usage which leads to a lower CPU cache consumption.
Chapter 7. CBR Case study

This also explains why actors perform better on expensive server CPU's with large CPU cache and large memory. In this context, it is worth to mention that the 50 million msg/sec on a single machine claim made on Akka actor website\(^3\) is based on a synthetic benchmark running on a 48 core Server\(^4\) with 128GB of memory. These fact suggest that actors tend to have higher hardware demands compared to other approaches.

For comparison, the disruptor is, just like actors, an event based approach, but performs several magnitudes better while consuming only a small fraction of the resources actors would require. However, actors are designed for distributed systems, whereas the disruptor does not directly supports distributed data processing, but requires third party message systems such as ZMQ\(^5\).

The third insight is the most surprising one: Functional concurrency in Scala can perform as good as the disruptor, which is known to be the fastest concurrency model available for Java. The underlying ring buffer of the disruptor is entirely based on eliminating CPU cache misses at any stage during program execution. Even though the performance and in particular the very low variance and latency are very appealing, the complex and verbose programming leaves the disruptor as a niche solution. Parallel collections and nested Futures, on the other hand, are very compact and simple to program. Out of the 126 LoC required to implement the CBR calculator using Futures nested in parallel collections, only 7 LoC are actually related to the used concurrency model. For the disruptor, the ratio reverses, which means, out of the 534 LoC, only 50 code lines are not related to the disruptor concurrency model. In terms of productivity, nested Futures and parallel collection deliver the best performance with the lowest number of code lines.

---

\(^3\)www.akka.io
\(^4\)www.letitcrash.com/post/20397701710/
\(^5\)www.zeromq.org/
7.6 Threats to validity

Validity of the benchmark results is approached by using excel for cross-validation of all values calculated by the benchmark. Cross-validation means, min, max, average, variance and standard derivation are calculated by corresponding excel functions on the same raw data and compared to the results calculated by the benchmark. However, due to a rounding issue in Excel, the cross validation for variance and standard derivation differs in Excel. I need to clarify that a simple check with a scientific calculator proves excel wrong and verifies the results calculated by the benchmark. This was not the intention of the cross-validation, but having at least min, max and mean values verified increases the confidence in the presented results. Also, all raw-data and spreadsheets are on the thesis website and can be further validated and analyzed with better statistical tools such as SPSS.

Internal validity could be affected by the case selection. While it is true that CBR was chosen because of the unique property of supporting task and data parallelism, it is not the general case for concurrency programming. Addressing this bias, all results are separated in SIMD and non-SIMD tables allowing different conclusions for task, data and combined parallelism. However, having just once case investigated is the biggest thread to internal validity. This is an acknowledge flaw in the experiment design. I argue, this flaw is acceptable because the purpose of the case study was not the investigation of how f-DBC performs for any possible scenario but instead only if or how it would affect concurrency in Scala. Therefore, investigating all available concurrency approaches on just one case study helps for comparing results. Nevertheless, a bigger sample surely will improve internal validity and remove the natural bias of a one-sample example.

Another aspect of internal validity is the question if or how much the CBR implementation was tailored for the benchmark. Even though performance is very important due to the high computational cost of case based reasoning, only generic performance optimization of known issues in Scala have been applied. Section 7.7 contains more details about applied performance optimization and the actual implementation details of the chosen SIMD data parallelism. Apart from that, no optimization specifically to CBR has been used.
Chapter 7. CBR Case study

For clarifying what CBR specific performance optimization means, a few more details of CBR system are required. First, domain specific knowledge is often used to apply task specific optimization to a CBR system. For instance, the first optimization would be the reduction of query results by excluding cases that are obviously dissimilar in order to reduce the number of comparisons. A second CBR specific optimization would be caching results from comparisons that occur frequently, in order to prevent recomputing the same scores again. A third optimization would be a reduction of attributes used for calculating the similarity score. By applying entropy determining algorithms, such as InformationGain, only those few attributes with the highest discrimination score could be used for calculating the similarity score required for the CBR algorithm. This approach of dimension reduction has a profound impact on the computational cost of the CBR algorithm, since a quarter of all attributes are often enough to separate over 95% of all cases.

However, for the CBR case study, none of these domain specific optimizations have been applied. Instead, the design of the benchmark reflects the worst possible scenario in a CBR system which means, all cases are compared to all others using all attributes for similarity calculating. This design was chosen by purpose to maximize the computational cost in order to measure the real impact of f-DBC on concurrency. In contrast, a fully optimized CBR system using cached computational results would arbitrary bias performance measurements.

**External validity** is affected to certain degree by the design of the experiment. The conclusion that mixing a contract in an implementation of a shared interface does not affect performance is valid and supported by the data. However, generalizing this conclusion by suggesting f-DBC would not affect application performance at all is not supported by the data. Furthermore, this conclusion is only valid for the chosen contract. A different, more complex, contract may cause a certain performance penalty. On the other hand, the conclusion that f-DBC as a framework does not add any other performance penalty than the one caused by the contract is valid. This conclusion leads to the question of how to design good contracts that do not degrade performance. One answer is the used contract and the corresponding interface that serves as a reference.
The second conclusion of SIMD data parallelism improving performance of all implementations with the exception of actors is true for the CBR use case, but cannot be generalized. As the CBR scenario was chosen because of its known support of data-parallelism, this decision limits the generalizability of the results. On the other hand, the performance results of performance improved by data parallelism are in line with findings by Totoo et al. published in 2012 [129]. Their findings indicate a similar disappointing result for actors. Their conclusions states:

The "lower-level actor-based code in Scala does not manage to improve performance significantly and therefore the extra programming effort is not justified in this version." (6 in [129]).

This verifies my results and, for the scope of SIMD data parallelism, allows the generalization of the conclusion, that actors are not a suitable choice for this kind of task.

Third, the conclusion of functional concurrency in Scala being able to perform as fast as the fastest concurrency in Java while only requiring one fifth of LoC cannot be generalized, as there is just one experiment to back this claim. Even though there are some real world projects\(^6\) reporting ten times better performance while reducing magnitudes of latency just by using the disruptor internally, there is no other comparison between the disruptor and functional concurrency in Scala. Therefore, more experiments with different kind of tasks needs to be done before any valid conclusion can be drawn.

\(^6\)http://logging.apache.org/log4j/2.x/
Conceptually cased based reasoning (CBR) is described by the CBR-cycle which comprises four activities

1. **Retrieve** similar cases to the problem description

2. **Reuse** a solution suggested by a similar case

3. **Revise** or adapt that solution to better fit the new problem

4. **Retain** the new solution once it has been confirmed or validated.

Case retrieval is further divided into querying existing cases and calculating a similarity score that defines how similar two cases are. For the scope of this thesis, I have implemented only the retrieve stage by loading all cases from a data file and calculating a similarity score based on all attributes. A travel case is implemented as a case class in Scala, as shown in listing 7.2.

```scala
final case class Case(journeyCode: Int, caseId: String,
  holidayType: HolidayType, price: Int,
  numberOfPersons: Short, region: String,
  country: Country, transportation: Transportation,
  duration: Short, season: Month,
  accommodation: Accommodation, hotel: Hotel)
```

*Listing 7.2: CBR travel case class*

The fields HolidayType, Transportation and Month are implemented as Scala Enum’s. A Scala Enum represents each entry with an numeric index, which is used for similarity calculation. The journey code is a unique identifier for each case.

For each of the twelve attributes, a corresponding local similarity function is defined. A local similarity score for an attribute calculates the distance of the actual value of from target to the source case. Taking the attribute `numberOfPersons` as an example, if the number of persons is identical in both cases, the similarity score is one. If the numbers differs, depending on how far the values are apart, a lower similarity score is calculated.
Listing 7.3 shows the implementation of the person similarity score. First, the similarity score is only calculated if the attribute is set (1b); if not, its value is set to zero (1b). Second, each calculated score is multiplied by the attribute weight to reflect the attributes relevancy correctly. Third, if the values of person attribute are too far apart, the score is set to zero in order to prevent an unnecessary bias of the global similarity score.

```java
protected def getPersonsSimilarity(refPersons: Short, c2Persons: Short,
   isSet: Boolean, w: Double): Double = {
  if (isSet) { // (1a)
    if (c2Persons == refPersons) ONE * w // (2)
    if (c2Persons == refPersons + 1 || c2Persons == refPersons - 1) 0.90 * w
    if (c2Persons == refPersons + 2 || c2Persons == refPersons - 2) 0.60 * w
    if (c2Persons == refPersons + 3 || c2Persons == refPersons - 3) 0.40 * w
    if (c2Persons == refPersons + 4 || c2Persons == refPersons - 4) 0.20 * w
    if (c2Persons == refPersons + 5 || c2Persons == refPersons - 5) 0.05 * w
    else ZERO (3)
  } else ZERO // (1b)
}
```

**Listing 7.3:** Local similarity score

The remaining eleven attributes follow the same approach of determining the similarity score on how far values are apart from each other. String attributes, such as a region name, have a simplified similarity function that only determines if two strings are identical or not. This is due to a shortcoming of the used data set, which has no ontology of regions. Modelling geographical distances in a local similarity calculation, however, is not practical because of the large number of global regions used in the data. As the String equality check is sensitive towards typographic errors, an automated data checking and correction has been performed to ensure all occurrences of all region names are spelled the same way in the data set.

The global similarity score, as shown in listing 7.4, sums all attribute similarity scores and divides the sum by the sum of all attribute weights.

The quality of a similarity function depends on how well the local similarity scores are defined. This enforces a trade-off between accuracy in the sense of an accurate distance metric that reflects the real difference between two cases and computational cost.
Chapter 7. CBR Case study

Listing 7.4: Global similarity score

The more accurate a similarity function gets, the more expensive the comparison between cases becomes, because calculating a more detailed scores requires more comparisons and attribute checks.

7.7.1 Optimization

In order to cope with the general high computational cost of case based reasoning, the following generic performance optimization have been applied:

- Scala HashMap is replaced by java.util.HashMap.
- Lazy val is used for invariant computational results.
- private[this] is used to inline fields and small methods.
- Closures are avoided.
- Lists are replaced with arrays.

Using Java’s HashMap as main data structure is known to provide at least two or three times faster reading performance\(^7\). The exact reason why Scala’s own HashMap performs that badly is not known to me, but I was able to measure the loss in performance. Lazy val is applied to store invariant computational results. This is, for instance, the case for the sum of all attribute weights. In order to keep object creation time unaffected, the sum is stored lazily, which means, the value is calculated by first access and then stored in the field.

\(^7\)http://capecoder.wordpress.com/2012/07/29/
Using private[this] supports compiler inlining, which means, a reference is replaced with the actual value for faster access. Closures in Scala are generated anonymous inner classes, which are known for rather poor performances. Avoiding Closures entirely, however, is difficult, as for-loops generate Closures prior to compilation. Apart from for-loops, Closures are not used in the CBR implementation. Lists in Scala are known to be a slow data-structure, therefore arrays and hashmaps are used instead.

These are all known best practices that apply to Scala and none of them are specific to case-based reasoning. Even though the chosen CBR case study implements a rather simple similarity function, computational cost is already very high. However, with applying those performance optimizations performance has improved noticeably in the sense of a two times lower execution time.

### 7.7.2 Data parallelism

For data-parallelism, SIMD has been chosen. SIMD refers to Single Instruction, Multiple Data and means, one specific computation instruction is applied to multiple data of the same kind. This approach is also known as Structure of Arrays\(^8\), because the main data structure consists of arrays.

Enabling efficient SIMD parallelism is possible because of the fully indexed data-set. Each case has a unique integer ID, which is used to decompose the used HashMap into an array of arrays. For each attribute, an array is created, containing all values for all cases. In order to keep track of which value belongs to which case, the case index is mapped to the array-index. Through that, local similarity is calculated on a batch of values. Furthermore, all twelve arrays of attributes can be calculated concurrently.

Next, the global similarity score is calculated by iterating over all resulting attribute arrays, summing up the values of each attribute array at the iterating index, divided by the sum of all weights, and stored in a resulting array. A HashMap is created by re-mapping array indices from the resulting array to case id’s.

\(^8\)http://hectorgon.blogspot.co.nz/2006/08/
The performance impact of SIMD data parallelism varies significantly depending on several factors. First of all, CPU cache size needs to be considered, since processing an array of data is only fast if the entire array fits into the CPU cache. Second, linear task execution usually benefits less from data parallelism compared to concurrent execution. Finally, benchmarks reflecting the actual use-case need to be done in order to determine the exact benefit of SIMD.
Chapter 8

Empirical DBC usage

This chapter presents firstly the empirical study how design by contract is used in Eiffel and in Scala. Then it presents a direct comparison of the two studies and finally analyzes the findings. All data and results of the study are available online\textsuperscript{1}.

8.1 Methodology and metrics

The goal is to identify the usage of design by contract by using two metrics. The first metric counts the number of occurrences for each of the DBC keywords in each source code file of that language. This task is performed with two custom bash scripts, one for counting DBC in Eiffel and another one for counting DBC in Scala. Both scripts are listed in Appendix B.

The second metric determines the source code size by counting lines of codes (LoC). For LoC counting, SCLC\textsuperscript{2} is used, as it is one of the few tools supporting Eiffel. Lines of codes excludes comments, blank lines and counts per token, which means, a single curly brace is counted as a line. The chosen metrics are calculated on a data set of 8 projects written in Eiffel and 12 projects written in Scala.

\textsuperscript{1}Available at: http://marvin-hansen.github.io/Msc/

\textsuperscript{2}https://code.google.com/p/sclc/
8.2 Data

All eight projects written in Eiffel were collected from the official subversion repository\(^3\), hosted on Eiffel.com. Code lines are measured as raw, counting all lines including blank lines and comments; as comments, only counting comments; and as code, counting code without blank lines and comments. All eight projects have approximately five million lines of code altogether, in which Eiffel Studio contributes the largest share to the code base. Table 8.1 summarizes the used data set for Eiffel.

<table>
<thead>
<tr>
<th>Project</th>
<th>Code</th>
<th>Comments</th>
<th>Raw</th>
</tr>
</thead>
<tbody>
<tr>
<td>aranea</td>
<td>9137</td>
<td>1225</td>
<td>12875</td>
</tr>
<tr>
<td>egigs</td>
<td>8744</td>
<td>854</td>
<td>11518</td>
</tr>
<tr>
<td>Eiffel-media</td>
<td>122433</td>
<td>21461</td>
<td>169524</td>
</tr>
<tr>
<td>EiffelStudio</td>
<td>3738059</td>
<td>464431</td>
<td>4764260</td>
</tr>
<tr>
<td>elogger</td>
<td>13921</td>
<td>3541</td>
<td>19889</td>
</tr>
<tr>
<td>goanna</td>
<td>47704</td>
<td>8233</td>
<td>63959</td>
</tr>
<tr>
<td>mec</td>
<td>1038499</td>
<td>118489</td>
<td>1312980</td>
</tr>
<tr>
<td>vampeer</td>
<td>67878</td>
<td>9235</td>
<td>84653</td>
</tr>
</tbody>
</table>

Table 8.1: LoC for 8 Eiffel projects

For Scala, twelve projects were chosen, having altogether 359.199 lines of code. As Scala is a relatively new programming language and has a very compact syntax compared to Eiffel, it is difficult to find projects with a large code base. Despite the lower LoC count, all chosen Scala projects are using DBC to a certain degree. The calculated LoC metrics for Scala are summarized in table 8.2.

\(^3\)https://svn.eiffel.com/
### 8.3 Results

For Eiffel, the results are skewed by the small sample size and the large contribution of Eiffel Studio. In order to balance the results, Eiffel Studio has been removed from result calculation, and the median value has been calculated additionally to the average. A direct comparison between Eiffel Studio, the median value and average value of the remaining seven projects clearly illustrates that only Eiffel Studio heavily uses design by contract. All other projects make less usage of postconditions and barely ever use invariant checks.

For Scala, none of the projects have skewed the data as heavily as Eiffel Studio for Eiffel, thus all Scala projects have been used for calculating median and average value. Unlike Eiffel, Scala does not have an invariant check by default, thus only requires (precondition) and ensuring (postcondition) are counted. Unlike Eiffel, Scala projects barely write postconditions at all and preconditions are even less used.
8.3.1 Eiffel

The results for DBC usage in Eiffel are summarized in table 8.3, which also contains median and average values. The comparison between median and average shows that the median value is closer to the actual observation per project. The median code size for all eight projects is 47.704 LoC with a median of 513 postconditions, 1167 preconditions and 57 invariant checks.

The main observation from this data set is, that precondition is the most used contract keyword, and invariant the least used one. Postcondition usage occurs frequently in some projects, such as mec and Eiffel-media, and barely in others, such as egigs.

<table>
<thead>
<tr>
<th>Project</th>
<th>LoC</th>
<th>Ensures</th>
<th>Requires</th>
<th>Invariant</th>
</tr>
</thead>
<tbody>
<tr>
<td>aranea</td>
<td>9137</td>
<td>123</td>
<td>116</td>
<td>40</td>
</tr>
<tr>
<td>egigs</td>
<td>8744</td>
<td>6</td>
<td>146</td>
<td>0</td>
</tr>
<tr>
<td>Eiffel-media</td>
<td>122433</td>
<td>1769</td>
<td>1741</td>
<td>269</td>
</tr>
<tr>
<td>elogger</td>
<td>13921</td>
<td>50</td>
<td>313</td>
<td>56</td>
</tr>
<tr>
<td>goanna</td>
<td>47704</td>
<td>513</td>
<td>1167</td>
<td>168</td>
</tr>
<tr>
<td>mec</td>
<td>1038499</td>
<td>11917</td>
<td>14822</td>
<td>2535</td>
</tr>
<tr>
<td>vampeer</td>
<td>67878</td>
<td>1367</td>
<td>1724</td>
<td>57</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>186.902</strong></td>
<td><strong>2249</strong></td>
<td><strong>2861</strong></td>
<td><strong>450</strong></td>
</tr>
<tr>
<td><strong>Median</strong></td>
<td><strong>47.704</strong></td>
<td><strong>513</strong></td>
<td><strong>1167</strong></td>
<td><strong>57</strong></td>
</tr>
</tbody>
</table>

Table 8.3: DBC usage for 7 Eiffel projects

Eiffel Studio uses contracts; in particular invariant checks (7.396) are used significantly more often compared to the median (57) of the remaining seven projects. To bring Eiffel Studio into perspective, table 8.4 shows the data for Eiffel Studio compared to the average, median, min and max value of the remaining seven projects.

Beginning with the observed LoC count, Eiffel Studio has three times more LoC compared to the maximum observed LoC of the remaining samples. The obvious conclusion would be that Eiffel Studio would have three times more contracts at maximum. Evidently, this is not the case. Instead of three times, four times more pre- and postconditions are observed in Eiffel Studio.
Table 8.4: DBC usage in Eiffel Studio

Only invariants are within the estimated usage range, which is particularly interesting when compared to the low average and median value. To illustrate this difference further, graphic 8.1 shows the plotted numbers, which makes it clear, how intense Eiffel Studio uses contracts compared to all other projects.

Figure 8.1: DBC usage in Eiffel Studio compared to median and average usage

Summarizing the results for Eiffel, apart from preconditions, contracts are barely written, with the exception of mec and Eiffel Studio.
8.3.2 Scala

Similar to the Eiffel sample, the overall data distribution of the Scala sample is uneven and skewed by two large projects (lift and breeze). Table 8.5 presents the results for DBC usage in Scala. Contracts are written in form of preconditions, even though the overall usage is considerably low.

The median usage is 59 preconditions per project, which is twenty times lower than the usage in Eiffel. Postcondition plays only an insignificant role in Scala. Out of all 12 projects, only 11 postconditions were counted in four projects. The remaining eight projects are not using postconditions at all.

<table>
<thead>
<tr>
<th>Project</th>
<th>LoC</th>
<th>Ensuring</th>
<th>Requires</th>
</tr>
</thead>
<tbody>
<tr>
<td>Akka</td>
<td>67262</td>
<td>4</td>
<td>227</td>
</tr>
<tr>
<td>Breeze</td>
<td>30125</td>
<td>0</td>
<td>673</td>
</tr>
<tr>
<td>kiama</td>
<td>21426</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>Lift</td>
<td>66650</td>
<td>0</td>
<td>502</td>
</tr>
<tr>
<td>Play2</td>
<td>35954</td>
<td>0</td>
<td>78</td>
</tr>
<tr>
<td>sbt</td>
<td>36542</td>
<td>1</td>
<td>82</td>
</tr>
<tr>
<td>scalabioalg</td>
<td>2190</td>
<td>0</td>
<td>50</td>
</tr>
<tr>
<td>scalaz</td>
<td>28914</td>
<td>5</td>
<td>16</td>
</tr>
<tr>
<td>shapeless</td>
<td>7900</td>
<td>0</td>
<td>69</td>
</tr>
<tr>
<td>slick</td>
<td>13189</td>
<td>0</td>
<td>17</td>
</tr>
<tr>
<td>spec2</td>
<td>31997</td>
<td>0</td>
<td>23</td>
</tr>
<tr>
<td>spire</td>
<td>17050</td>
<td>1</td>
<td>22</td>
</tr>
<tr>
<td>Average</td>
<td>29933</td>
<td>1</td>
<td>147</td>
</tr>
<tr>
<td>Median</td>
<td>29519</td>
<td>0</td>
<td>59</td>
</tr>
</tbody>
</table>

Table 8.5: DBC usage in 12 Scala projects

In general, the results for Scala show very clearly, that postconditions are not used, regardless of the project size. Preconditions, on the other hand, tend to correlate with project size, as Akka, Lift and Breeze have above median and average preconditions. Figure 8.2 plots the data from table 8.5 to make the data distribution visible.

Play2 and Lift are both popular web frameworks for Scala, but only Lift uses a noticeable number of preconditions. Play2, despite having half the LoC, has seven times less preconditions.
This observation makes it clear, that the number of written contracts varies significantly even amongst projects that share the same usage scenario. Another observation is that frameworks heavily using generics as abstraction, such as spire\textsuperscript{4} and scalaz\textsuperscript{5} tend to use less contracts compared to the median value of the sample.

\textbf{Figure 8.2:} DBC usage for 12 Scala projects compared to median and average usage

\textsuperscript{4}Spire is generic numeric library providing structures for algebra and category theory
\textsuperscript{5}ScalaZ is provides generic type Classes and purely functional data structures for Scala,
8.3.3 Eiffel to Scala comparison

Comparing the usage of pre- and postcondition in Eiffel\(^6\) to Scala shows a major difference in how contracts are used in both languages. As summarized in table 8.6 and 8.7, Scala projects have an order of magnitude less preconditions and almost insignificant postconditions compared to Eiffel.

| Requires | Scala | Eiffel |
|----------|-------|--------|          |
| Median   | 59    | 1167   |          |
| Average  | 147   | 2861   |          |
| Min      | 7     | 116    |          |
| Max      | 673   | 14822  |          |

Table 8.6: Precondition

<table>
<thead>
<tr>
<th>Ensures</th>
<th>Scala</th>
<th>Eiffel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Median</td>
<td>0</td>
<td>513</td>
</tr>
<tr>
<td>Average</td>
<td>1</td>
<td>2249</td>
</tr>
<tr>
<td>Min</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>Max</td>
<td>5</td>
<td>11917</td>
</tr>
</tbody>
</table>

Table 8.7: Postcondition

To put the low level of specification in Scala into perspective, the median LoC for projects of the Scala sample is 29519 LoC, but only 59 (median) preconditions are used, which is equivalent to one precondition per 500 LoC.

The seven Eiffel projects, on the other hand, have a median LoC of 47,704 with 1167 (median) preconditions, which translates to roughly one precondition per 40 LoC. In other words, in terms of preconditions, Eiffel is over twelve times more specified compared to Scala.

8.4 Analysis

Why do Scala and Eiffel differ so much in terms of written contracts?

Compared to Scala, Eiffel is a verbose programming language, which means, assertions are required to a certain degree, as the large amount of source code can hardly be fully analyzed manually. Scala, on the other hand, has a concise syntax combined with a compact programming mode that follows best practices from functional programming. This means, in Scala functions are simple to reason about and require less assertions compared to Eiffel. If the correctness of a function can be determined by simple reasoning, there is no need to place a contract.

\(^6\)Eiffel Studio is excluded from the Eiffel data to minimize the bias
Conceptually, Eiffel is an object oriented programming language, thus it relies on certain established practices, such as shared state and mutable fields, which are known to be error prone in concurrent execution. From all Eiffel projects, only Eiffel Studio is known to use concurrency, which the large number of invariant checks confirm. Invariant checks are required to ensure a mutable field has not changed if accessed from multiple threads. Scala, on the other hand, does not have invariant checks, as functional programming style with immutable fields is preferred for concurrency programming in Scala.

One question is, whether the large number of contracts written in Eiffel can be assumed to be correct. As the ongoing research effort in Eiffel shows, a large proportion of effort is related to automated contract inference because of largely incomplete or incorrect contracts\(^7\). The quantity of contracts in Eiffel apparently does not necessarily lead to better software in terms of less testing, but equally less to good contracts. Scala, on the other hand, is conceptually an object-functional language known for its very compact syntax, which leads to an overall lower LoC. There is no evidence of the quality of contracts written in Scala, but as the Eiffel research shows, this is mainly an issue of postconditions. As Scala developers hardly write any postconditions, this particularity is not an issue. Instead, the next obvious question is, why are Scala developers writing so few pre-conditions and almost no postconditions?

It appears to me, Scala’s sophisticated type system is more used to define certain constraints on functions, which would be declared as a precondition otherwise. More precisely, type variance and generalized type constraints in Scala allowing further specification of a generic type, even down to one single specific type. Through that, there is very little usage left for preconditions in generic programming in Scala. The down-side of Scala’s powerful type system and compiler support is a certain degree of complexity, which is a controversial topic amongst Scala developers\(^8\).

The absence of postconditions in Scala is most likely because caused by best practices from functional programming. For instance, one best practice in functional programming is mutating function parameters only while using only immutable fields within a function. Combined with the convention that the last statement in a function (or branch) is always the return value, a developer can reason very quickly about the

\(^7\)see section 3.2 for details

\(^8\)http://yz.mit.edu/wp/true-scala-complexity/
correctness of a function just by checking if only parameters are mutated, immutable fields are used and if the last statement reflects the intended return value. If all those criteria are fulfilled, there is no reason to place a post-condition in a function.

In summary, it appears Scala’s lower demand for contracts is mainly due to best practices from functional programming and good support from the type system. Regardless of the underlying reasons, the direct consequence of evidently missing preconditions in Scala is the nullification of the assumption contracts that could be used as test-oracles for automated testing. Postconditions are of particular interest, because they define an exact specification of the expected output for a method or function. Since the assumption of existing postcondition evidently is not confirmed in practice, a different approach is required. This insight has lead to the investigation of compiler support for concurrency programming in Scala, as presented in the next chapter.
Chapter 9

Scalarius: Concurrency macro

This chapter presents Scalarius, a macro for the Scala compiler that verifies concurrency correctness by preventing all known side-effects in a function from compilation. A function without any side-effect is also known as a pure function. Pure functions are important because they are thread-safe by definition\(^1\). Even though functional style is preferred for concurrency programming in Scala, surprisingly, there is no direct compiler support available during development. Scalarius checks for all known side-effects and issues compiler errors whenever a side-effect is detected in a function guarded by the macro.

9.1 Usage

Using the macro requires the sbt\(^2\) build tool in order to include the macro as dependency in a project. Details on how to configure SBT are available online\(^3\).

The actual usage of the macro only requires the new keyword *pure* to be placed in front of the function body. For demonstrating the macro, I use a function that calculates the absolute value of the sum of three arrays. Because summing values is an associative\(^4\) operation the sums for all three arrays can be calculated concurrently. The absolute value on the other hand is not associative so it has to be applied to the final sum of all arrays.

---

\(^1\)See section 4.9 for details

\(^2\)http://www.scala-sbt.org/

\(^3\)http://www.scala-sbt.org/0.12.3/docs/Detailed-Topics/Macro-Projects.html

\(^4\)\(f(a + b) + g(c) = = f(a) + g(b + c)\)
Chapter 9. Scalarius: Concurrency macro

Listing 9.1 shows a trait that contains the basic data structure in the form of a case class (1) a non-blocking concurrency implementation (2), a purely functional implementations of the absolute sum, and the basic functions (3) used in both implementation.

The concurrent implementation uses for-comprehension (see 4.10), on futures (see 4.11.5) in order to compute the sum for each array independently (2a). The absolute function is then called on the sum of all sums(2b) before the final result is returned(2c).

All functions are guarded by the `pure` macro (p) preventing any side effects from compilation. Once this code compiles, it is proven to be thread-safe because of the verified absence of all side effects that could affect thread safety.

```scala
import src.main.scala.CheckMacro.pure

trait pureFunctions{

  // (1)
  case class measurements(sampleA : Array[Int],
                           sampleB : Array[Int],
                           sampleC : Array[Int])

  // (2)
  def absSumFut(m : measurements): Int = pure {
    // (p)
    val f = for {
      f1 <- Future(sum(m.sampleA))  // (2a)
      f2 <- Future(sum(m.sampleB))
      f3 <- Future(sum(m.sampleC))
    } yield abs(f1 + f2 + f3)  // (2b)
    Await.result(f, 1 second)  // (2c)
  } // (3)  // (p)

  def abs(a: Int): Int = pure { if (a < 0) a * -1 else a}

  def sum(l: Array[Int]) = pure {l.sum}
}
```

**Listing 9.1:** macro: pure functions

Applying those pure functions to a set of data is shown in listing 9.2. The main method contains the side effect of console output which makes it impure thus the macro is not applied to it
Chapter 9. Scalarius: Concurrency macro

```
object main extends pureFunctions {
  def main(args: Array[String]){
    val ml = measurements(Array(1,2,-3,4), Array(-5,6,7,8),
      Array(9,-10,11,12))
    println("[Futures] Abs value of all sums is: " + absSumFut(ml))
  }
} //Output:
> [Futures] Abs value of all sums is: 42
```

**Listing 9.2: macro: usage example**

This code compiles because function purity is not violated. However, if any side-effect occurs within a pure function, the macro issues a compiler warning. For example, re-writing the absolute function for using a mutable variable as return value, as shown in listing 9.3, issues a compiler warning.

```
def abs(a: Int): Int = pure {
  var ret : Int = if (a < 0) a * -1 else a
  return ret
} // compile error:
```

**Listing 9.3: macro: compiler warning**

The macro not only prevents mutable fields in pure functions, but also all remaining side-effects, as summarized in the Scala background chapter in section 4.9. Listing 9.4 shows an example that contains all covered categories of side-effects and the corresponding compiler error messages. As the example shows, the error message for each side effect contains a brief explanation or advice on how to resolve the reported issue. The important observation is that the macro covers several overloaded methods, such as `readLine` which has multiple implementations for different types.
import src.main.scala.CheckMacro.pure

def impureTest = pure {
  lazy val impureX = null
  // (1) throws: null is disabled. Use Option[T] if the return value is optional
  val impureY = ???
  // (2) throws: Inferred type containing Nothing from assignment.
  // Avoid empty list and empty statements such as ???
  var impureZ = 4
  // (3) throws: var is disabled. Use val instead
  val impureR = readLine()
  // throws: Console reading is disabled. Use IO monad instead
  println("Won’t print")
  // (4) throws: Console output is disabled. If required, use impure function instead
  val impureW1: Writer = new PrintWriter(new File("test.txt"))
  // (5) throws: File writing is disabled. Use IO monad instead.
  val impureFile = Source.fromFile("test.txt")
  // (6) throws: File reading is disabled. Use IO monad instead
  throw new Exception
  // (7) throws: Exception throwing is disabled. Use Option[T] and getOrElse instead
  }

Listing 9.4: macro: usage example

Even though side effects such as null(1), nothing(2) dereferences or mutable (3) fields are important in concurrent execution, once prevented by the compiler, sequential code also improves. The same applies to all file or network input or output operations which should be generally centralized in corresponding classes. By applying the macro to all functions and methods that process data, thread-safety and concurrent execution is fully guarded by the compiler. There are some cases when a method may or may not return a result.

In such a case, instead of throwing an exception, the corresponding error message (7) suggests using Option[T] instead for runtime handling of non-results. The difference is that the application does not terminate in the case of an unexpected result but instead, getOrElse applies an alternative in the case of an unexpected return value.
9.2  Approach

The main difficulty in concurrency programming is thread-safety, the assurance that
parts of a program can be be used simultaneously by different threads. Address-
ing this issue, I have chosen to develop a macro for the Scala compiler that verifies
common pitfalls in functional programming and applies equally to concurrency pro-
gramming in Scala. Apart from all known side-effects (see 4.9) the macro checks
additionally for null and nothing references to prevent NullPointerException exceptions at
run-time. Essentially, the macro guards a pure function by enforcing restrictions
such as:

- disables mutable fields and data structures
- disables incorrectly declared Unit (void in Java) return type.
- disables null values and references
- disables type nothing
- disables empty implementations
- disables non initialized values
- disables console input and output
- disables file reading and writing
- disables exceptions

Even though this macro is just a small step towards compiler verified concurrency,
it already supports developer actively by preventing unsafe code from compilation.
Without this level of compiler verification, a function cannot be explicitly known to
be pure which means thread-safety is not guaranteed.
9.3 Compile checks

9.3.1 Null and mutable check

```scala
def varTest() = pure {
  var x = 4    // (A1)
  var y = null // (A2)
  var l = List(1,2,3) // (A3)
}
Listing 9.5: macro: var check
```

```scala
def nullTest() = pure {
  lazy val x = null    // (B1)
  val y = ???         // (B2)
  val l = List.empty   // (B3)
}
Listing 9.6: macro: null check
```

Mutable fields are a well known side effect and a source of common thread safety issues. Disabling mutable fields, as shown in listing 9.5, applies to all value types (A1), null references (A2) and collections such as lists (A3). The implementation of the mutable checks searches for occurrences of the `var` keyword which implies any custom type or class, is guarded as well by this check.

Null reference checks in Scala require a closer look because Scala defines five different null values (see 4.3) but only `null`, `Nothing` and `Nil` are directly accessible during programming. The remaining two are abstract types. The macro covers all three accessible null types (listing 9.6) which means each of them triggers a compiler error referring to the actual location of the issue. One impotent difference from all other null checks in Scala is that the `lazy` keyword (B1) for late evaluation does not prevent the detection of null references. This is a small but significant difference because late evaluation means a value only gets assigned or computed by first access.

The problem is that, a normal not-null check written in Scala cannot evaluate lazy values because the value is not accessible prior to the first access after compilation. Consequently, any lazy value passes normal compilation even if its assigned to null. Once the code is running and a lazy value set to null gets accessed the first time, the application terminates with a NullPointerException. The main motivation for including a null check in the compiler macro is because only a compile check can detect null references on lazy values.

The triple question mark in (B2) indicates a function or value that has not yet been implemented or assigned. This is convenient for compiling unit tests that are not implemented but it should not be used in pure functions.
Chapter 9. Scalarius: Concurrency macro

The type of the triple question mark is type *Nothing* and is therefore detected by the macro. The type of empty list(B3) is defined as *Nil* which is an alias to type *List[Nothing]* which is also detected by the macro.

These different meanings of *null* and *Nothing* are important because, the inference of type *Nothing* can equally lead to runtime exceptions but for different reasons. In order to separate errors caused by *null* and *Nothing*, the macro issues two different compiler errors (1)(2) as shown in listing 9.7.

```scala
1 [error] /path/to/src.scala:32: null is disabled
2 [error] lazy val x = null // (1)
3 [error] /path/to/src.scala:15: Inferred type containing Nothing from assignment
4 [error] val y = ??? // (2)
```

**Listing 9.7:** macro: reported errors

### 9.3.2 Unit check

The next category of checks provided by the macro are *Unit* checks that detect orphaned statements by checking first whether they have a specified return type and if not, if the return type is *Unit*. If a statement has neither, it is reported by the Unit check. Preventing statements that look like *Unit* but return something else has the benefit of detecting orphaned statements which are often the source of difficult to debug bugs.

To illustrate the issue, listing 9.8 uses the absolute value (*abs*) as example. The abs function (1) is defined as the non-negative integer value. A simple `abs(-42)` (2) will be reported as an error (4) by the macro because, unlike the first impression of having no return value, the actual return value is an integer but somewhat hidden. In other words, even though the absolute value is calculated correctly, it is not used. Assigning the return value to a (immutable) field, as shown in the correct example (3) prevents this error.
Chapter 9. Scalarius: Concurrency macro

Debugging those incorrectly used statements in a larger algorithm is can be difficult because the \( \text{abs}(-42) \) statement suggests that it has no return value (Unit) but in fact, it has one which is not obvious from reading the code as example (2) shows.

```scala
1  def returnUnitTest = pure {
2    // (1)
3    def abs(a : Int) : Int = {if (a < 0) a * -1 else a}
4    abs(42)  // (2) Wrong: Statements must return Unit
5    val a = abs(42)  // (3) Right: Val assignment returns Unit!
6  }  // (4) Compile error:
7  [error] /path/to/src.scala:35: Statements must return Unit
8  [error]     abs(42)
9  [error]
```

Listing 9.8: macro: UnitChecks

9.3.3 IO checks

One of the key attributes of pure functions is the absence of IO operations. Therefore, the macro prevents all major input and output methods available in Scala.

```scala
1  def inTest() = pure { 1  def outTest() = pure {
2    val a = readLong()  // (A1) 2    println("Doesn’t print")  // (B1)
3    val b = Source.fromFile("")) // (A2) 3    val w2 = new BufferedWriter()  // (B2)
4    val c = new DataInputStream()  // (A3) 4    val w3 = new FileOutputStream()  // (B3)
5  }  // (A4) Compile error:
```

Listing 9.9: macro: Input checks  Listing 9.10: macro: Output checks

Beginning with input operations, listing 9.9 shows a small set of guarded input methods such as console reading (A1), file reading (A2) or reading from a network resource. The macro all specialized reading methods, for instance it detects all fourteen different methods to read specific types such as Int, Float or Double. Furthermore, not only are DataInputStream are detected but all defined InputStreams. Output detection follows a similar pattern; the macro detects console printing (B1), file writing (B2) and any kind of OutputStream(3). Also, the macro is not limited to the examples shown in listing 9.10, but also covers all three console print and write methods defined in Scala.
9.3.4 Exception check

Even though Scala supports all exceptions defined in Java, \textit{Option}[/T] is the preferred way to deal with unexpected return values. An option contains either \textit{Some}[/T] which is the expected result of type \textit{T} or \textit{None} which is the unexpected result. An unexpected result can be either an empty value or null. As null is already disabled through the macro, leaving \textit{None} by its default empty value is the preferred way of using option. By convention, an option shifts the exception handling to the caller which can retry or switch to a different function call to manage an unexpected result. Since \textit{Option}[/T] is a practical alternative way of handling unexpected return values, exception throwing can be safely disabled in pure functions.

Apart from detecting all JVM default exceptions, the macro also detects all custom written exceptions regardless of whether the defining class is written in Scala or Java. Listing 9.11 shows a custom (A) and an non-catchable\(^5\) exception (A1) used in a test function (B).

```scala
class ChuckNorrisException extends Exception // (A)
class JulesWinnfieldException extends Exception{sys.exit(42)} // (A1)
def exceptionTest(i : Int) = pure {
  if (i < 2) { new Exception} // (B)
  else if (i < 42) { throw new NumberFormatException}
  else if (i > 23) { throw new ChuckNorrisException}
  else { throw new JulesWinnfieldException}
}
```

Listing 9.11: macro: Exception checks

The macro detects (C) all custom exceptions in a guarded function regardless of whether they can be thrown or even reached or not.

\(^5\)http://stackoverflow.com/questions/13883166/
9.4 Implementation

Implementing a macro for the Scala compiler requires a definition of a new keyword, and an implementing class the defines what action the Scala compiler performs on each occurrence of the new keyword. The definition of the new keyword `pure` is shown in listing 9.12. First, the macros need to be imported in order to access the compiler API. For Scala 2.10, macros are marked as experimental\(^{(1a)}\) but the upcoming 2.11 release is considered to contain a stable macro implementation\(^6\). Additionally reflection support is\(^{(1b)}\) required as well in order to query type informations during compilations.

An object is used as a container for the macro\(^{(1)}\) itself in order to ensure only singleton instances of the macro are accessed by the compiler. Writing a macro uses a macro definition, which represents the facade of the macro. A macro definition is a normal function but its body is a reference to an implementation. By convention, type T in the signature of a macro definition must correspond to a parameter of type `c.Expr[T]` in the signature of a macro implementation. In listing 9.12, the macro definition `pure`\(^{(3)}\) takes as parameter an expression of type T. Type T refers to the actual type of the function body associated with the `pure` macro. The implementation `threadSafeImpl`\(^{(4)}\) then follows convention by only using `c.Expr[T]` in its signature. The used context c is generated by the compiler.

```scala
1  import language.experimental.macros    // (1a)
2  import scala.reflect.macros.Context    // (1b)
3  object CheckMacro {    // (2)
4    def pure[T](expr: T) = macro threadSafeImpl[T]    // (3)
5    // (4)
7    import c.universe._
8    ....
```

**Listing 9.12:** macro: keyword definition

The macro implementation contains a set of methods that all follow the pattern of traversing the abstract syntax tree (AST) which is accessible through the compiler API. Traversing the AST is used for searching particular term using pattern matching and issues a custom error message whenever the term is found.

\(^6\)http://java.dzone.com/articles/state-scala-2013
This process is illustrated on the null check implementation in listing 9.13. First, each check is implemented as an object that inherits from the `Traverser` class. Second, the traverse method needs to be overridden with a custom implementation. Third, pattern matching is used and whenever a particular term, in this case `null` is matched, a custom action can be performed. In the listing, an error is thrown by using the context variable `c` (provides the actual compiling context) to query the exact position of the match and define a custom error message. The second case uses the underscore arrow notion which means all other (underscore) cases will be ignored (single arrow). Next, the `traverse` method inherited by the super class `Traverse` is called in order to trigger the actual AST traversing. Finally, the defined check is initiated by a static call to the overridden traverse method.

```-scalarius
/** check for null references */
object NoNull extends Traverser {  //1
    override def traverse(tree: Tree) {  //2
        tree match {
            case Literal(Constant(null)) =>  //3
                c.error(tree.pos, "null is disabled")  //4
            case _ =>  //5
                super.traverse(tree)  //6
        }
    }
    NoNull.traverse(expr.tree) //7
}

Listing 9.13: macro: null check implementation
```

The remaining checks such as `NoVar` are implemented using the same approach of traversing the AST and apply pattern matching on each expression.

Only IO and exception checks use a different approach. During AST traversal, expressions are passed to a check function that performs a sub-type relation check using generic type constraints\(^7\). The original intention of generic type constraints is to constraints generic type parameters on method scope. However, the upper bound conformance test \(A <: B\) is implemented as an attempt to type-cast from type \(A\) to \(B\) and returns true if that attempts succeeds or false otherwise. Using this upper bound conformance test in a slightly different way allows an efficient test to determine if a particular expression has a specific super-type, for instance `Exception`.

\(^7\)See section 4.5.4 for details
Because of the strict inheritance hierarchy of all exceptions any custom exception needs to implement either super-type Exception in order to be throwable. Conversely, a type-cast from any custom type to super-type Exception will succeed and therefore pass the upper bound conformance test. Listing 9.14 shows the implementation of the exception test. Once called from the AST traversal method, the checkExceptions takes a sub-tree (1) as parameter and traverses this tree further for each expression(1). The sub-type check (3) then tests whether type the expression has Exception as upper type bound or not. The importance of this particular detail needs to be stressed as neither Scala nor Java have any kind of super-type test defined. Only this particular type bound test allows detection of any kind of (custom) exception with one single test. If the type bound test succeeds, a Boolean flag is set to true and finally triggers a compiler error (1) showing the position of the expression and a custom error message.

```scala
1  def checkExceptions(statements: Tree){  // (1)
2    statements.foreach {  // (2)
3      stat =>  // (3)
4        val isExc: Boolean = stat.tpe <: typeOf[Exception]
5        if (isExc) { (4)
6          c.error(stat.pos, excErrMsg)
7        }
8      }
9    }
```

Listing 9.14: macro: exception check implementation

The same approach is used to detect all implementations of input or output streams, readers or writers. Through that, the macro detects the usage of countless specific IO classes with only five tests all-together. Furthermore, expanding the macro only requires definition of a new sub-type test and another category of classes would be reported by the compiler macro.
9.5 Limitations

The first and obvious limitation of the macro is its limited application scope to functional concurrency. This is not necessarily a limitation by itself because Scala puts strong emphasis on functional concurrency and the overall good performance together with the very compact syntax makes it a favourable choice of non-functional style such as actors.

The second limitation is its low level of supporting empirical evidence. Because of the limited time available during this thesis, the macro has been fully implemented and tested but no empirical study of its impact on concurrency programming has been made.

The third and less obvious limitation of using the macro is a longer compile time compared to compiling without the macro. In particular, excessive checking of a large number of functions slows the Scala compiler down. The exact impact needs to be quantified but there is a perceived reduction in compilation speed. This is most likely because of its very early development stage without any performance optimization. Future work may quantify the exact performance loss and improves performance of the macro.
9.6 Analysis

The macro divides functions into proven to be pure functions and all others assumed to be impure due to the lack of verification. In a concurrency context, requiring only pure functions guarantees proven thread-safety by default. There is no need for any testing or further verification effort since the macro prevents any impure functions from compilation.

The macro verification is also in line with best practice in functional programming using a pure core and a thin IO layers that handles IO operations and user interactions.

In larger projects, verifying the pureness of the core is difficult due to the high number of functions involved. On the other hand, using the macro for verification ensures that function purity, and therefore, thread-safety is proven to correct after compilation.

The obvious benefit of compiler verification is the elimination of all concurrency related testing. Concurrent execution testing is not required any more because the compiler has already proven that all issues affecting concurrent execution are absent. Because of this proof, testing concurrent code is not required any more which increases productivity during development.
Chapter 10

Discussion

The idea of using contracts as test oracles for automated testing is appealing because re-using a well-known concept in a different context may advance the long standing issue of reducing testing effort.

I did an analysis of design by contract in Eiffel, in Ada, in Scala and finally reviewed 15 DBC projects in Java. Based on this result, I developed f-DBC, a new type of contract framework. Unlike existing work, f-DBC (Chapter 6) allows profile and alias based configuration per package, per class or even per method. Similar to Ada, f-DBC supports functions as contracts but unlike Eiffel, f-DBC does not enforce system termination in case of a contract violation. Instead, f-DBC allows flexible, profile based violation handling, such as logging, email notification or even custom action.

The first question was: How does f-DBC affect run-time performance of an application? In a comprehensive case study (Chapter 7), I evaluated the performance impact on five different concurrency implementations in Scala as well as a Java based implementation of the disruptor, which is known to deliver the fastest possible concurrency performance in Java. The result is threefold. First, the impact f-DBC has on run-time is insignificant and within the measurement tolerance. Second, parallel collection and nested futures, which are both the highest level of abstraction in Scala’s concurrency, perform as fast as the disruptor. Third, parallel collection, nested futures and the disruptor scale efficiently with the number of cores.
Chapter 10. Discussion

Based on these results, the next question was: If tool support is not an issue any more, would contracts be written in practice? The evidence provided in chapter 8 makes it clear neither Eiffel nor Scala projects are using required postconditions, thus generating test oracles from contracts cannot be done in practice. One remaining question is: Why Scala developers do not writing contracts?

I can only suggest a few thoughts on this observation because I do not have any supporting evidence such as survey results. From my own experience in writing considerably complex Scala code, the main difference to other languages, such as Java or C#, is the simplicity to reason about correctness. The combination of functional style and a powerful type system already cover the majority of issues I would otherwise use a contract for. Another observation is that I only write preconditions and never postconditions. This is in line with the empirical results I have collected from other projects. This observation is mainly based on one minor shortcoming in Scala’s type system that does not allow constraints on type values similar to ADA’s dynamic predicates. Apart from this minor issue\(^1\), there is no reason to write a contract in Scala.

The answer to the question, whether design by contract could be used as foundation for automated testing, is no, DBC is not a suitable choice. This answer is based on the presented evidence of postconditions not being written in Scala. Even if contracts are written, as the evidence from Eiffel projects suggests, the quality is clearly not sufficient, as the ongoing research\(^2\) on inferring more complete and correct contracts shows. Therefore, the underlying assumption of transforming a postcondition into a test-oracle is invalidated. The next question was: Is there an alternative way for verifying correctness concurrency code?

As the evidence from the case study shows, functional concurrency correctly programmed can be in some cases as fast as the fastest known concurrency in Java. However, verifying correctness of functional concurrency is equivalent to verifying functional purity, since pure functions are thread safe by definition.

---

\(^1\)Constraints on type values could be added with some work on the typer of the Scala compiler

\(^2\)Section 3.2 in background chapter
Following this thought, I have implemented a macro (Chapter 9) for the Scala compiler that introduces a `pure` keyword which prevents all known side effects from compilation. If a function contains any side effect, for instance a mutable field, the compiler reports the precise location of the occurrence with an explanation of the cause, so that the issue can be resolve instantly.

The direct consequence of this strict verification is that no testing is required for the guarded code to ensure its correctness in any concurrent execution model. Unlike complex contracts, there is zero impact on application performance. Even though f-DBC checks do not have any measurable performance impact, complex contracts may have. Compiler checks on the other hand never affect the final application performance, regardless of the number or complexity of the checks.

Compiler performance is affected by the macro, but the issue of slow compiling Scala code has already been known for a long time. Because of that, there are several solutions, such as incremental compilations, the FSC (Fast Scala Compiler) and continuous compile servers which already reduce compilation time in Scala. It needs to be evaluated how macro usage can be accelerated by these compiler acceleration techniques.

A valid criticism on the results is the validation approach. The insignificant performance impact of f-DBC could be caused by a tailored contract that fits the example. The observation of Scala developers not writing postconditions could be limited to the small sample of just twelve projects. Also, the claim of the compiler macro preventing all known side effects was made without an empirical study as evidence.

While all these aspects are true, there are three considerations I suggest to be taken into account. First, the CBR case study was not meant to evaluate the performance impact of one specific contract but the overall additional computational cost of performing contract checks by the f-DBC framework. From this perspective, the low computational cost f-DBC causes by adding checks remains valid. What indeed needs to be evaluated is, whether this observation changes with a large number of contracts. I suggest to do such an evaluation as subject of further work because the topic of performance impact depending on contract size is not related to this thesis. Second, I suggest that increasing the sample size for investigating DBC usage in Scala would not affect the observation of general low usage of contracts in Scala.
Some of the most prominent Scala projects such as Play, Lift and Akka are already in the sample and none of them uses postconditions. Many of those projects are running large scale production systems, which means if writing postconditions had any benefit, it would have been applied in those projects. I argue: If the most prominent and widely used Scala projects, which are maintained by skilled and experienced developers, are not using postconditions, it is even less likely that smaller and less known projects would use them.

Without doubt, there are some exceptions, for instance, out of all twelve projects, only two projects are using together just 11 postconditions. These two outliers actually back my argument, because these two projects, ScalaZ and Akka, are known to be maintained by highly skilled and experienced developers. This means, at least in these two projects, developers are evidently aware of DBC, but they still do not write more contracts. Therefore, increasing the sample size would not necessarily change the observation of the general low usage of contracts in Scala.

Third, the compiler macro was only evaluated on so called "fail-examples", which are examples tailored to trigger a specific compile error. It is true that there is no supporting empirical study verifying any of the claims made about the effectiveness of the macro. I suggest, such a study is not required because of the way the compiler macro has been implemented.

Taking the exception checking of the macro as one example, the implementation performs a sub-type relation check on any term of a function AST, which means there is no loophole left to circumvent the compiler check. I need to emphasise that this kind of sub-type relation check is frequently used in the macro implementation. This means, there is very little use of a large scale empirical study evaluating whether all side effects are reported because these effects are detected by sub-type relation checks catching any kind of a side effect by definition.

Instead, I suggest either a formal proof or an external code review for ensuring the correctness of the macro. This would be sufficient to provide solid evidence for the claim made that all known side effects are detected by the macro. However, due to the limited time and resources during this thesis as well as the early experimental stage of Scala macros, I suggest such a correctness evaluation of the macro is subject to further work. In particular once Scala macros are considered as stable, which is planned for early 2014.
Chapter 11

Future work

“I never think of the future - it comes soon enough.”

*Albert Einstein, 1879 – 1955*

11.1 Overview

Distributed programming has emerged from a specialized niche towards a broader adoption due to the rise of cloud computing. Cloud computing means moving applications away from the desktop into web applications. These applications and files are hosted on distributed systems consisting of thousands of computers and servers, all linked together and accessible via internet. With the ongoing exponential rise of smart phones and the popularity of mobile broadband, internet services for mobile devices are required to scale in new dimensions by using cloud technology [130][131][132].

However, there is no clear consensus on what exactly the term ”cloud computing” means. Even though the vague definition of data center provided service over the internet tells us what it could be, there is by no means a particular paradigm or technology implied. Instead, the term ”cloud” is an umbrella term, referring to a broad variety of distributed concepts, such as grid Computing, cluster computing and distributed systems in general [133]. The only generally accepted core component of cloud computing is distributed programming, which is known to be challenging.
Chapter 11. Future work

Traditional low level distributed programming based on C/C++ and FORTRAN through Message Passing Interface\(^1\) (MPI) is complex and difficult to accomplish in a reasonable time. The main source of complexity in programming distributed systems is dealing with system faults and interrupted or delayed network connections \([134]\). Because of the need to build highly scalable cloud systems in a short time, new concepts, such as map reduce, actors and more recently distributed data flow, have emerged. Despite some progress in large scale data analytic and real-time message processing, general distributed programming still remains challenging. Approaching the challenge of reducing complexity in distributed programming while preserving high performance in a fault tolerant way, I propose a novel approach of distributed functional concurrency, using elastic fault prevention.

The novelty of the approach is threefold. First, it is based on automatically converting any pure function into a remote function by using compiler code re-writing. Second, it uses automatic remote deployment and a generic code execution that is able to swap-in and execute any function on a node during run-time. Third, elastic fault prevention scales automatically with the number of faults by dispatching tasks from non-responding systems to working ones automatically. Instead of defining complex error handling manually, elastic fault prevention monitors nodes autonomously, restarts non responding ones and dispatches jobs automatically as required. Therefore, the configuration effort can be as little as providing a list of fall-back nodes or as big as providing custom handling strategies for different kinds of faults and network segments.

\(^1\)www.open-mpi.org/
11.2 Background

Distributed programming aims to compute complex tasks simultaneously on different computers. Distributed programming is divided into three categories:

1. Parallel algorithms in a shared-memory model

2. Parallel algorithms in a message-passing model

3. Distributed algorithms in a message-passing model

Parallel algorithms with shared memory assume that all computers have access to a shared memory. A practical approach of shared memory is asynchronous shared memory, which uses Compare And Swap (CAS) operations synchronizing a shared distributed memory. Shared memory generally suffers from problems such as mutex or deadlocks, and requires sensitive access to the shared state.

Parallel algorithms in a message-passing model use messages to communicate between computer nodes. The algorithm and a specific network topology can be chosen for implementation [135]. Distributed algorithms in message passing model means all computers run the same program, but communicate over messages only regardless of the underlying network. One practical example of distributed algorithms in a message passing model are actors\(^2\). An actor encapsulates an algorithm in strict shared nothing environment and communicates only by passing immutable messages between actors. An actor system, however, is agnostic of the underlying network topology or protocol.

11.3 Problem

There are many challenges involved in distributed systems, but the main complexity of distributed programming emerges from three areas [134]:

1. Adverse cooperative computing

2. Routing

3. Linear approximation

\(^2\)see section 4.11 in Scala chapter
Chapter 11. Future work

One of the fundamental challenges in distributed computing is performing the computation of tasks in a decentralized setting, where the computing medium is subject to adversarial perturbations. Essentially, distributing computation and data requires failure handling that obscures the original simple computation with large amounts of complex code in order to deal with possible issues. For instance, implementing an actor system requires a significant effort to provide fall-back and recovery strategies for software, hardware and network failures. Coping with failures remains an open challenge in distributed programming (6 in [134]) [136].

Routing data in distributed systems depends on congestion and stretch, as well as the extra delay of data distribution in a congestion free network. Ideally, congestion and stretch should be minimal, but in practice routing is required to cope with network congestion. The open problem in distributed routing is the identification of circumstances in which congestion and stretch can be optimized simultaneously [134].

Integer programs are a particular kind of optimization in which some or all of the variables are restricted to integer encoding. Integer-linear programming reflects the reformulation of a complex problem to an equivalent integer program that has linear run time performance [137]. Approximating distributed programs to the performance of general integer-linear programs is a problem that remains unsolved [134].

11.4 State of the art

Related works in distributed programming are distributed data flow, map-reduce and actors. Distributed data flow extends data-flow [138] towards distributed computing on streams (flow) of data. Map reduce is a distributed batch processing approach, aiming at simpler programming of large scala offline data processing. Map reduce is based on the concept of transferring the algorithm to the stored data, processing them locally and returning only the computational result, which then will be aggregated with other results processed in parallel [136]. The strength of map reduce is large scale data analytic; for instance, data mining on user data, but its weakness is real-time data processing. Even though with some tweaks applied map reduce can perform close to real-time, it cannot reach full real-time because of the underlying batch-job architecture fully relying on job schedulers.
Chapter 11. Future work

Actors\(^3\), on the other hand, are designed for real-time message processing \([119]\). The strength of actor systems is fault-tolerant high throughput processing, but its weakness are computational-intense tasks. The CBR case study in chapter 7 illustrates on a computational demanding task how sub-standard actors perform compared to other approaches. Apparently, the gain of high availability architecture comes at the price of reduced performance and efficiency, which is mostly due to the underlying complex programming model.

Distributed data flow is currently under active research and not used in practice, mainly because of its very early stage. Even though the concept of distributed data flow was first introduced in 2003 \([139]\) and first applied to web service composition \([139]\), research on a solid theoretical foundation only started in 2009 \([140]\) \([140]\).

Despite important contributions, such as *Live Distributed Objects* \([141]\) and scalable protocols via distributed data flows \([142]\), only BLOOM\(^4\), a research programming language based on Ruby implements distributed data flow \([143]\). Bloom is a data-centric programming language for disorderly distributed programming with consistency analysis based on eventually consistency. The central concept of BLOOM is based on the monotonic property, which is, a hypothesis of any derived fact may be freely extended with additional assumptions. In programming, monotonicity means, any extension in output needs to be derived of additional input. Consequently, any retraction is considered to be non-monotonic.

BLOOM’s consistency analysis is based on eventual consistency, which is based on proving logical monotonicity. Eventual consistency means, logically monotonic distributed code is consistent at a point in the future without any need for coordination protocols, such as locks or two-phase commit. Eventual consistency can be fully guaranteed by protecting non-monotonic statements with coordination protocols \([143]\).

The central aspect is the guarantee of the monotonic property and the strictly ordered execution of non-monotonic statements, which preserves data integrity, regardless of temporary non-determinism or computational interruptions. This is a relaxed criterion compared to ACID, but reflects the practical necessity for availability and responsiveness over total consistency in large scale internet systems \([143]\).

---

\(^3\)Actors are covered in section 4.11 in the Scala chapter

\(^4\)www.bloom-lang.net/
Chapter 11. Future work

However, as BLOOM is based on Ruby, it inherits Ruby’s dynamic type system and byte code interpretation, which causes security and performance issues. In particular, Ruby’s approach of ”Duck Typing”, that is guessing a type at run-time by only checking if a specific method is present, is problematic because it allows many ways to break those run-time checks. Section 4.4.7 in the Scala chapter explains the implied issues in more detail. Furthermore, as a dynamic language, byte code interpretation is required to preserve dynamic execution, but the flip side is a performance reduction. For many programming tasks, these two issues are not particularly important, but for distributed programming security and performance they are relevant, because distributed systems are meant to perform interruption-free high performance computation.

In summary, the related work is divided into three categories. The first one is large scale distributed batch data processing using map reduce. The second one is real-time message processing using actors, and the third one would be processing of data streams using distributed data-flow. The gap between all three categories is a general purpose distributed programming model that is very fast and simple to program correctly while efficiently utilising modern multi-core systems.

11.5 Proposal

Instead of inventing a new programming language, I propose a novel approach for reducing complexity of distributed programming in Scala:

\[ \text{Distributed functional concurrency (DFC)} \]

Distributed functional concurrency means, pure functions that are verified to be thread safe are deployed remotely in a distributed system and executed concurrently on each node. The underlying idea is to combine functional concurrency with a distribution model to accomplish a very high performance and low latency distributed computing model that is very fast, robust and efficient to program.

The difference between DFC to parallel algorithms is the absence of any assumptions considering the memory model, which does not require CAS operations for synchronization. The difference to message passing is that raw data are distributed without the overhead of a messaging system.
Similar to distributed algorithms, DFC is also network agnostic. Unlike distributed algorithms, no arbitrary encapsulation, such as actors, are required to distribute an algorithm. Instead, any pure function is shared over network. Unlike distributed data-flow in BLOOM, no proves of eventual consistency are required, because combined local consistency is used instead. Local consistency uses compiler support to verify consistency execution locally for each distributed function only. Global consistency is the aggregated result from all local consistency checks. Only if a valid global consistency state can be verified, a DFC program compiles. This is a significantly stricter consistency check compared to eventual consistency, as it does not make any arbitrary assumptions of a possible execution state in the future. Conceptually, distributed functional concurrency is:

1. a very compact programming model
2. distributes pure function
3. distributes raw data
4. compiler verified consistency

DFC borrows from nested futures, parallel collections and for-comprehension, and extends these concepts towards distributing functions. DFC pushes a function over network to a remote host that performs an anonymous invocation of the function for all received data. This invocation pushes the data to the underlying concurrency execution pool, which means, any data from a remote host are computed concurrently. This is motivated by keeping latency low for real time data processing. Once a result is computed, the remote host pushes the result back to the source that aggregates all results from all remote hosts.

To illustrate the concept further, listing 11.1 shows a proposed syntax based on the code example from the compiler macro in chapter 9. There are several important observations. First, the general concept of using functional composition, such as futures and for-comprehension (4), combined with case classes (1) remains the same. Second, for simplicity, remote hosts are defined as socket (3). Third, for defining a distributed (4) work-flow, a new keyword DFC (4a) is proposed. The DFC keyword is intended as a compiler macro for transforming pure functions into remotely executed functions.
Chapter 11. Future work

Listing 11.1: DFC proposed syntax

Fourth, each pure function is passed to a Remote, which essentially returns a future from a remote host that is passed as a second parameter (4b) back to the Remote wrapper. Conceptually, a Remote re-writes a function to a future that awaits a result from a remotely deployed instance of the passed function. This is the central and most significant language construct in DFC. Instead of writing complex structures for explicit remote execution, as actors would require, a Remote performs an implicit conversion from any pure function to a future of a remote function instance. This re-write required a macro that defines the Remote keyword.

Fifth, because a remote returns a future, functional composition is fully preserved allowing the usual aggregation (4c) of different futures. Finally, a DFC composition relies on function purity, which means, every used function (5) has is verified to be pure. The given example in listing 11.1 reflects the high level of abstraction accomplished by distributed functional programming.
Technically, DFC concurrency can be implemented in Scala by using:

1. Implicit object serialization
2. Implicit bidirectional network transfer
3. Remote function execution
4. Ring buffer based elastic fault prevention

### 11.5.1 Implicit object serialization

Scala has a unified object model\(^5\), which means that functions are values and each value is an object. Since a function is an object in Scala, it can be serialized and transported over network. Depending on the chosen approach, object serialization can be an expensive computational task impacting the throughput of a distributed system.

In order to achieve the fastest possible DFC throughput, I suggest investigating Scala pickling\(^6\), that is a fast, customizable and virtually code free object serialization for Scala. Experiments\(^7\) suggest that pickler combinators outperform state-of-the-art serialization frameworks and significantly reduce memory usage. Scala pickling is based on macros and implicit conversions developed at the École Polytechnique Fédérale de Lausanne (EPFL)\(^8\). Listing 11.2 shows a minimal working example that illustrates a simpler object serialization to json. Apart from json, a binary format for

```
1 trait picklerTest{
2  import scala.pickling._
3  import json._
4  // serialize to json
5  val pckl = List(1, 2, 3, 4).pickle
6  // deserialize back to Integer list
7  val lst = pckl.unpickle[List[Int]]
8 }
```

**Listing 11.2:** Scala pickling example

\(^5\)For details, see section 4.3 in the background chapter
\(^6\)http://lampwww.epfl.ch/~hmiller/pickling/
\(^7\)http://lampwww.epfl.ch/~hmiller/pickling/benchmarks/
\(^8\)http://lampwww.epfl.ch
Chapter 11. Future work

Scala objects is also supported. Alternatively, custom serialization formats can be added to Scala pickling as well. Performance evaluations are required to determine which approach provides best serialization performance.

11.5.2 Implicit bidirectional network transfer

One of the most limiting factors in distributed systems is network throughput. Instead of using traditional tcp/ip connections, I suggest the instigation of protocols that are optimized for bidirectional network communication, such as web sockets or Web RTC. Experiments\(^9\) show that web sockets can provide between a 500:1 to 1000:1 reduction in unnecessary HTTP header traffic and 3:1 reduction in latency. Similar to implicit data serialization, I suggest to define network transfer implicitly through a macro by generating static methods for each Remote wrapper in order to preserve a high level of performance.

11.5.3 Remote function execution

Higher ordered functions\(^10\) are functions that take another function as parameter. Using a higher ordered function together with generics\(^11\) allows passing an arbitrary function to another function for execution. Listing 11.3 shows a generic code runner implemented in Scala. There are two important details. First, the type parameters [T,R] define the function type T and the return type R of the function. The function f is passed by value, which is indicated by the val keyword. Second, the run function (2) takes input of type T, applies it to the function itself and returns a result of type R.

\begin{verbatim}
1 class CodeRunner[T, R] (val f : (T) => R) { // (1)
2   def run(input : T) : R = {f(input) } // (2)
3 }
\end{verbatim}

Listing 11.3: Scala code runner

---

\(^9\)urlhttp://www.websocket.org/quantum.html
\(^10\)For details, see section 4.9.2 in the Scala chapter
\(^11\)For details about generics, see section 4.4.2 in the Scala chapter
Chapter 11. Future work

Using the generic code runner requires just three steps, as illustrated in listing 11.4. First, a function (1) needs to be defined. Second, an instance of the code runner needs to be created and the function is passed as a parameter (2) to the code runner instance. Third, the run method is called with the actual function parameter (3), which will be passed to the function and finally return the result.

```
object Sandbox {
  def main(args: Array[String]) {
    // (1)
    def sum(l: Array[Int]) = {l.sum} // (1)
    // (2)
    val cr = new CodeRunner(sum) // (2)
    // (3)
    val res = cr.run(Array(1, 2, 3, 5, 12)) // (3)
    println("Result is: " + res) // (3)
  } //>Result is: 23
}
```

Listing 11.4: Scala code runner usage example

The generic code runner is a core concept in distributing functions, since it allows a very efficient remote execution of functions.

Bringing all three concepts together, I suggest to implement distributed Scala concurrency by writing a macro that re-writes the `Remote` keyword into a future connected to a socket. Next, the actual function is serialized in a lightweight binary object format and sent through a web sockets to a remote host. A remote host runs a minimal web socket server listing for incoming objects. For simplicity, only three different kinds of objects are allowed: function, data and control sequences. An incoming function is then deserialized and used to create new instance of a code runner. Once a function is remotely deployed, all following data are passed to the instance of the code runner, which executes the function on the incoming data.
Chapter 11. Future work

Each incoming data triggers a concurrent code execution. Concurrency has been excluded from the code runner example in listing 11.3 because experiments are required in order to determine which kind of concurrency provides the best performance on remote code execution. The results of the case study in chapter 7 are a good starting point for further exploration. The result from the code runner is then serialized and sends back to the source, which will return the result to the corresponding future for further processing.

11.5.4 Elastic fault prevention

Using a bi-directional connection based on web-sockets allows using control sequences to send commands like ping, start, stop or restart to a remote host. This helps to re-start a remote host, in case a software problem occurred. Also, if a ping fails, a DFC can exclude the remote host from further computations. These control sequences form the foundation for elastic fault prevention, which monitors nodes and takes action as required. The general concept of using connection control sequences for handling software faults needs further investigation and some experiments in order to determine its potential.

The general idea is twofold. The first one is a combination of connection monitoring and a set of control sequences handling common issues, such as software faults that can be fixed by restarting and node failures that can be fixed by dispatching associated tasks to another node. The second one is a broader change in the underlying architecture. Inspired by the high throughput and low latency from the disruptor framework by using a ring-buffer, I suggest a similar mechanism to handle network request. Instead of using a one to one connection, meaning one remote function gets deployed to just one remote host, a remote function gets deployed to many nodes connected through a ring buffer, which can be configured in different ways such as:

- First-wins (low latency)
- Scheduling (load balancing)
- Fall-back (high availability)
Assigning each remote function to a group of nodes interconnected with an asynchronous, lock-free\textsuperscript{12} ring-buffer allows a virtually bottleneck-free scaling of latency, availability and load balancing.

The first-win strategy means, the remote function is deployed to all nodes in a group and data are sent to all nodes for processing. This may look inefficient, but there are two important implications. First, if a node goes down, it happens without any side-effect. Second, as all nodes reply, the first win strategy ensures always the fastest possible processing even under very high system load. In order to avoid network congestion, each sub-group needs to be isolated in a (virtual) sub-net. The first win approach is ideal for ultra low latency high volume applications, such as real-time stock market analysis. The down-side is a higher number of required nodes for each remote function.

The scheduling configuration also deploys a remote function to all nodes, but sends data only to one node at a time. The ring-buffer is used to allow multiple nodes to claim one to n incoming data and only returns processes the claimed ones. This approach is best understood as an "inversion" because traditional network schedulers dispatch data by sending them to a node. The ring-buffer, however, lets the node pull the data required for processing. One particularity of the ring-buffer allows a node to read "batches" of data which means, if the ring-buffer contains a sequence of unclaimed data, the first pulling node can read all available data in one row instead of reading each data entry one by one. Once a batch is pulled, the node processes the entire batch and returns results in the moment they are available.

The benefit is, if a node takes longer for processing, it does not pull any new data until the previous ones are returned, which implies that the other nodes are pulling data instead. The strength of this configuration is an ideal utilization of all available nodes at the cost of lower latency, since a blocking node holds results back until the entire data batch is processed. This is ideal for very high work load that is not latency sensitive but requires full computational power. Scientific computer simulations would one example for this kind of work-load.

Also, as each remote function can be assigned to an individual group of nodes, computational resources can be fine grained assigned according to how demanding a particular remote function in practice is.

\textsuperscript{12}A one producer set-up is always lock free, since only one producer can write to the ring buffer
Chapter 11. Future work

By assigning only a few nodes to "cheap" functions, more nodes are exclusively available to functions that required intense computational power. The down-side of this strategy is, if a node fails, the pulled data are lost and need to be re-sent to the ring-buffer. As failure detection is expensive, this particular strategy needs further research in how to combine scheduling with first-win to get the best of both approaches.

The third one, the fall-back strategy, means, a number of nodes is available on spare and are dynamically assigned to a group of nodes. This can be combined with the previous two strategies to cope with spikes of system load. This is also an important aspect in dynamically re-balancing computational resources based on certain metrics such as latency, throughput or number of failed nodes. Essentially, the core concept is, to have a resource pool of available nodes that are automatically assigned to a remote function if a certain criterion is met. For instance, if a remote function looses more than two nodes, a new one from the resource pool is assigned. Copying with spikes follows the same idea. For instance, if a remote function drops in latency, this affects instantly the system, because of the aggregating for-comprehension suspenses until all remote functions have returned a future. In this case, adding some additional nodes to that remote function brings latency up again before the slowdown propagates further.

As already mentioned, all these concepts need further research and experiments in order to determine their practical application. Even though applying a ring-buffer to a node-interconnection is a minor technicality, the concept of elastic fault prevention built on top of it requires significantly more work and is one of the areas of exploration.

11.6 Areas of exploration

The proposed distributed functional concurrency has three areas of exploration. First, a complete implementation needs to be written and evaluated. The proposed concepts of implicit object serialization, implicit network transfer and remote function execution are a starting point for further exploration. Additionally, implementing the concept of elastic fault prevention requires further work in order to determine its practicality.
This may lead to new insights related to language integrated support for dealing with the problems of adverse cooperative computing and routing.

Second, DFC offers a unique opportunity to experiment with new solutions to the problem of linear approximation. Linear approximation is of particular interest because with an highly efficient remote concurrency implementation, the actual limiting factor in throughput performance would be the physical network connection. A performance evaluation on a high-speed network link may provide more insights on how close distributed functional concurrency could come to linear integer performance.

Third, a specific application as case study is another promising area to explore. Distributed functional concurrency offers the potential to simplify the fast development of distributed high performance domain specific solutions in Scala.

One particular field of interest is computational neuroscience, since simulations of the human brain are a use case well known for very high computational demand. However, until very recently, with only few exceptions, each neuroscience simulation application offering the option of a domain-specific programming language (DSL) came with its own proprietary language, specific to that simulator [144]. The problem is, neurological simulations cannot be exchanged unless a simulator is proving the functionality and all participants are using the same simulator.

Recently (2006), new domain specific programming languages such as NeuroQL [145], emerged to simplify data queries for large neurological databases. Conceptually, NeuroQL is database agnostic and generates standard SQL for a variety of supported databases. For neurological computing however, only recent (2009) effort [144] was made for creating a first domain specific programming language based on Python. However, because of its dynamic typing, Python is not necessarily a good choice for concurrency and distributed computing. Additionally, in Where’s the Real Bottleneck in Scientific Computing? Wilson summarized the main problem as:

"Increasingly, the real limit on what computational scientists can accomplish is how quickly and reliably they can translate their ideas into working code."

In the growing\(^{13}\) field of computational neuroscience, Investigating distributed concurrency in Scala would be one direction to translate new ideas quickly and reliably into high performing code.

\(^{13}\)In April 2013, the U.S. Gov. granted 100 Mil. USD on brain research
Chapter 12

Conclusion

The conclusion of this thesis is threefold. First, f-DBC improves the flexibility of development without decreasing application performance. One particularly important observation is the fact that parallel collections and nested futures only require one fifth of LoC of the disruptor or actors while providing a similar level of performance with a significantly lower development effort and time. Second, the presented evidence suggests that only a few preconditions but postconditions are not written in Scala. Third, the presented compiler macro prevents all known side effects in functions and therefore supports the development of thread-safe concurrency applications.

While working on this thesis, I’ve made three major experiences. First, I learned to work with Scala since I’ve never used that language before. The high level of details and underlying theoretical foundation feels intimidating at first sight, but over time, most of Scala’s concepts turned out to be well thought through and understandable. Second, I’ve learned several new ways to write concurrency code. In particular functional concurrency, such as futures and for-comprehension, is very appealing due to its compact syntax that allows very fast programming. Third, the major insight was that compiler programming correctly used replaced a significant amount of testing effort that would have been required without compile checks. Since the underlying motivation (chapter 2) of this thesis is entirely based on exploring new ways of verifying concurrency correctness. One way to approach this challenge would be automated testing of concurrent code.
As it turned out, using a macro that prevents any impure function from compiling replaces testing of concurrent code because verified to be pure functions are thread-safe. All results have been discussed and criticized in chapter 10. Even though there are some shortcomings in validating the results presented in this thesis, I believe this does not affect the main observation that compiler support simplifies concurrency verification.

The main contribution of this thesis is an in-depth investigation of verifying the correctness of concurrency code in Scala. The presented macro is a novel approach of compile-time concurrency verification for functional concurrency.

The importance of functional concurrency is twofold. First, it is a very efficient and fast way to program many core systems, regardless of the number or CPU’s, APU’s or cores in a system. Second, correctly done, functional concurrency in Scala is as fast as the fastest concurrency in Java, but requires only a fraction of lines of code. Finally, functional concurrency scales well in terms of improving performance with the addition of cores.

In this context, the importance of compiler checking cannot be underestimated as modern compilers have grown into powerful verification tools. For instance, during this thesis, one topic that re-occurred frequently was the significance of macros and implicits by defining entirely new programming approaches. For instance, implicit macros that convert case classes into monads or virtually code free object serialization by using implicit isomorphism generated by macros, to name just two recent examples. Furthermore, new approaches based on implicits and macros appear almost monthly on the Scala mailing list.

Also, with the growing power of the compiler API and in particular of the type checker, there is a certain possibility that compilers will be used more frequently to perform formal verification of program correctness. This may not solve the problem of performing complete proofs from a proven to be correct proofing system, but it makes the entire concept of formal verification at least more accessible to a broader audience. In particular, with the ongoing effort to simplify the complexity of macro programming while preserving the expressiveness, the barrier compiler programming is being lowered. A macro for verifying functional correctness is just one small step. Future work of expanding the concept of compiler verified correctness into other areas, such as distributed programming, would be the next step.
Appendix A

Appendix: The MIT License (MIT)

Copyright ©2013 <MarvinHansen>

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED AS IS, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
Appendix B

Appendix: DBC usage script

Listing B.1: Script for counting DBC usage in Scala

```bash
#!/bin/bash
find . -type f -name "*.scala" | xargs -n1 grep "require" -c | 
sed "s/0//" | sed '/^$/d' > require_count.txt |
```

Listing B.2: Script for counting DBC usage in Eiffel

```bash
#!/bin/bash
find . -type f -name "*.e" | xargs -n1 grep "ensure" -c | 
sed "s/0//" | sed '/^$/d' > ensure_count.txt |
```

```bash
sed "s/0//" | sed '/^$/d' > invariant_count.txt |
```

```bash
sed "s/0//" | sed '/^$/d' > require_count.txt |
```
Appendix C

Appendix: CBR contract

```scala
trait alias extends fDBC
trait ReasonerSpec extends CaseReasonerI with ConfigManager with alias {
  val _DBC_CHEKS = getDbcFlag

  abstract override def getMostSimilarCase(refCase: Case, cm: CaseManager, weights: FeatureWeights): Case = {
    if (_DBC_CHEKS) {
      dbc.precondition("Reference case must not be null", refCase != null)
      dbc.precondition("Case manager must not be null", cm != null)
    }
    super.getMostSimilarCase(refCase, cm, weights)
  }

  abstract override def getMostSimilarCases(nrCases: Int, refCase: Case, cm: CaseManager, weights: FeatureWeights): CaseMap = {
    if (_DBC_CHEKS) {
      dbc.precondition("Nr. cases needs to be at least one", nrCases > 1)
      dbc.precondition("Reference case must not be null", refCase != null)
      dbc.precondition("Case manager must not be null", cm != null)
    }
    super.getMostSimilarCases(nrCases, refCase, cm, weights)
  }
}
```

Listing C.1: f-DBC CBR contract
Bibliography


Bibliography


Bibliography


Bibliography


Bibliography
