

# Fundamental Haskell notes

Anton Latukha

February 26, 2020

# Contents

<b>I</b>	<b>Introduction</b>	<b>21</b>
<b>II</b>	<b>Definitions</b>	<b>24</b>
<b>1</b>	<b>Algebra</b>	<b>25</b>
1.1	*	25
1.2	Algebraic	25
1.3	Algebraic structure	25
1.3.1	*	26
1.3.2	Fundamental theorem of algebra	26
1.4	Modular arithmetic	26
1.4.1	*	26
1.4.2	Modulus	26
1.4.2.1	*	26
<b>2</b>	<b>Bind</b>	<b>27</b>
2.1	*	27
<b>3</b>	<b>Category theory</b>	<b>28</b>
3.1	*	28
3.2	Abelian category	29
3.2.1	*	29
3.3	Composition	30
3.3.1	*	30
3.4	Endofunctor category	30
3.5	Functor	30
3.5.1	*	31
3.5.2	Power set functor	31
3.5.2.1	*	31
3.5.2.2	Power set functor laws	31
3.5.2.3	Lift	32
3.5.2.4	Power set functor is a free monad	32
3.5.3	Functorial	32
3.5.4	Forgetful functor	32
3.5.4.1	*	32
3.5.5	Identity functor	32
3.5.6	Endofunctor	33
3.5.6.1	*	33
3.5.7	Applicative functor	33
3.5.7.1	*	33
3.5.7.2	Applicative law	34
3.5.7.3	*	34
3.5.7.4	Applicative function	34
3.5.7.5	Special applicatives	35

3.5.7.6	Monad	36
3.5.7.7	Alternative type class	49
3.5.7.8	$\llangle \leq * \geq \rrangle$	50
3.5.8	Monoidal functor	50
3.5.9	Fusion	50
3.5.10	$\llangle \leq \$ \geq \rrangle$	50
3.5.11	Multifunctor	50
3.5.11.1	*	51
3.5.12	*	51
3.6	Hask category	51
3.6.1	*	51
3.7	Magma	51
3.7.1	Mag category	52
3.7.1.1	*	52
3.7.2	Semigroup	52
3.7.2.1	*	52
3.7.2.2	Monoid	52
3.8	Morphism	54
3.8.1	*	54
3.8.2	Homomorphism	54
3.8.2.1	*	54
3.8.3	Identity morphism	54
3.8.3.1	Identity	55
3.8.3.2	Identity function	55
3.8.4	Monomorphism	55
3.8.4.1	*	55
3.8.5	Epimorphism	56
3.8.5.1	*	56
3.8.6	Isomorphism	56
3.8.6.1	*	56
3.8.6.2	Lax	56
3.8.7	Endomorphism	57
3.8.7.1	Automorphism	57
3.8.7.2	*	57
3.8.8	Catamorphism	57
3.8.8.1	*	57
3.8.8.2	Catamorphism law	57
3.8.8.3	Anamorphism	58
3.8.9	Kernel	58
3.8.9.1	Kernel homomorphism	58
3.9	Set category	59
3.10	Natural transformation	59
3.10.1	*	60
3.10.2	Natural transformation component	60
3.10.2.1	*	60
3.10.3	Natural transformation in Haskell	60
3.10.4	Cat category	61
3.10.4.1	*	61
3.10.4.2	Bicategory	61
3.11	Category dual	61
3.11.1	Coalgebra	62
3.12	Thin category	62
3.12.1	*	62
3.13	Commuting diagram	62
3.13.1	*	62

3.14	Universal construction	62
3.14.1	*	63
3.15	Product	63
3.15.1	*	63
3.16	Coproduct	63
3.16.1	*	64
3.17	Free object	64
3.18	Internal category	64
3.19	Hom set	64
3.19.1	*	64
3.19.2	Hom-functor	64
3.19.3	Exponential object	65
3.19.3.1	*	65
3.19.3.2	Enriched category	65
<b>4</b>	<b>Data type</b>	<b>66</b>
4.1	*	66
4.2	Actual type	66
4.3	Algebraic data type	66
4.3.1	*	66
4.4	Cardinality	66
4.4.1	*	66
4.5	Data constant	67
4.6	Data constructor	67
4.7	data declaration	67
4.8	Dependent type	67
4.8.1	*	67
4.9	Gen type	67
4.10	Higher-kinded data type	67
4.10.1	*	67
4.11	newtype declaration	68
4.12	Principal type	68
4.13	Product data type	68
4.13.1	*	68
4.13.2	Sequence	68
4.13.2.1	*	69
4.13.2.2	List	69
4.14	Proxy type	69
4.15	Static typing	70
4.16	Structural type	70
4.16.1	*	70
4.17	Structural type system	70
4.17.1	*	70
4.18	Sum data type	70
4.19	Type alias	70
4.20	Type class	70
4.20.1	*	70
4.20.2	Arbitrary type class	71
4.20.2.1	Arbitrary function	71
4.20.3	CoArbitrary type class	71
4.20.3.1	*	71
4.20.4	Typeable type class	71
4.20.4.1	*	71
4.20.5	Type class inheritance	71
4.20.6	Derived instance	72

4.20.6.1 *	72
4.21 Type constant	72
4.22 Type constructor	73
4.23 type declaration	73
4.24 Typed hole	73
4.24.1 *	73
4.25 Type inference	73
4.25.1 *	73
4.26 Type class instance	73
4.27 Type rank	73
4.27.1 *	74
4.28 Type variable	74
4.29 Unlifted type	74
4.29.1 *	74
4.30 Data structure	75
4.30.1 Cons cell	75
4.30.2 Construct	75
4.30.2.1 *	75
4.30.3 Leaf	75
4.30.4 Node	75
4.31 Linear type	75
4.31.1 *	75
4.32 NonEmpty list data type	75
4.33 Session type	76
4.34 Binary tree	76
4.35 Bottom value	76
4.35.1 *	76
4.36 Bound	76
4.36.1 *	76
4.37 Constructor	76
4.37.1 *	76
4.38 Context	77
4.38.1 *	77
4.39 Inhabit	77
4.40 Maybe	77
4.40.0.1 *	77
4.41 Expected type	77
4.42 ADT	77
4.43 Concrete type	77
4.44 Type punning	78
4.45 Kind	78
4.45.1 *	78
4.46 IO	78
<b>5 Declaration</b>	<b>79</b>
<b>6 Differential operator</b>	<b>80</b>
6.1 *	80
<b>7 Dispatch</b>	<b>81</b>
<b>8 Effect</b>	<b>82</b>
<b>9 Evaluation</b>	<b>83</b>

<b>10 Expression</b>	<b>84</b>
10.1 *	84
10.2 Closed-form expression	84
10.3 RHS	84
10.4 LHS	84
10.5 Redex	85
10.6 Concatenate	85
10.7 Alpha equivalence	85
10.8 Ground expression	85
10.8.1 *	85
10.9 Variable	85
10.9.1 *	85
<b>11 First-class</b>	<b>86</b>
<b>12 Function</b>	<b>87</b>
12.1 *	87
12.2 Arity	88
12.3 Bijection	88
12.3.1 *	88
12.4 Combinator	88
12.5 Function application	89
12.5.1 *	89
12.6 Function body	89
12.7 Function composition	89
12.7.1 *	89
12.8 Function head	89
12.9 Function range	90
12.10 Higher-order function	90
12.10.1 *	90
12.10.2 Fold	90
12.11 Injection	90
12.11.1 *	91
12.12 Partial function	91
12.13 Purity	91
12.13.1 *	91
12.14 Pure function	91
12.15 Sectioning	91
12.16 Surjection	91
12.16.1 *	92
12.17 Unsafe function	92
12.17.1 *	92
12.18 Variadic	92
12.19 Domain	92
12.20 Codomain	92
12.21 Open formula	92
12.22 Recursion	92
12.22.1 *	92
12.22.2 Base case	93
12.22.3 Tail recursion	93
12.22.4 Polymorphic recursion	93
12.22.4.1 *	93
12.23 Free variable	93
12.24 Closure	93
12.24.1 *	93

12.25	Parameter	93
12.25.1	*	94
12.26	Partial application	94
12.26.1	*	94
12.27	Well-formed formula	94
12.27.1	*	94
<b>13</b>	<b>Homotopy</b>	<b>95</b>
13.1	*	95
<b>14</b>	<b>Lambda calculus</b>	<b>96</b>
14.1	*	96
14.2	Lambda cube	96
14.2.1	*	97
14.3	Lambda function	97
14.3.1	*	97
14.3.2	Anonymous lambda function	97
14.3.2.1	*	97
14.4	$\beta$ -reduction	97
14.4.1	*	97
14.4.2	$\beta$ -normal form	97
14.4.2.1	*	98
14.5	Calculus of constructions	98
14.5.1	*	98
14.6	Curry–Howard correspondence	98
14.6.1	*	98
14.7	Currying	98
14.7.1	*	99
14.8	Hindley–Milner type system	99
14.8.1	*	99
14.9	Reduction	99
14.9.1	*	99
14.10	$\beta$ - $\eta$ normal form	99
14.10.1	*	99
14.11	$\eta$ -abstraction	99
14.11.1	*	100
14.12	Lambda expression	100
<b>15</b>	<b>Lense</b>	<b>101</b>
<b>16</b>	<b>Operation</b>	<b>102</b>
16.1	Constant	102
16.2	Binary operation	102
16.2.1	*	102
16.3	Operator	102
16.3.1	Shift operator	102
16.3.1.1	*	102
16.4	Infix	103
16.5	Fixity	103
16.5.1	*	103
16.6	Zero	103
<b>17</b>	<b>Permutation</b>	<b>104</b>
<b>18</b>	<b>Phrase</b>	<b>105</b>

<b>19 Point-free</b>	<b>106</b>
19.1 *	106
19.2 Blackbird	106
19.2.1 *	106
19.3 Swing	107
19.4 Squish	107
<b>20 Polymorphism</b>	<b>108</b>
20.1 *	108
20.2 Levy polymorphism	108
20.3 Parametric polymorphism	108
20.3.1 Rank-1 polymorphism	108
20.3.1.1 *	108
20.3.2 Let-bound polymorphism	109
20.3.3 Constrained polymorphism	109
20.3.3.1 Ad hoc polymorphism	109
20.3.4 Impredicative polymorphism	109
20.3.4.1 *	109
20.3.5 Higher-rank polymorphism	109
20.3.5.1 *	110
20.4 Subtype polymorphism	110
20.5 Row polymorphism	110
20.6 Kind polymorphism	110
20.7 Linearity polymorphism	111
<b>21 Pragma</b>	<b>112</b>
21.1 LANGUAGE pragma	112
21.1.1 LANGUAGE option	112
21.1.1.1 *	112
21.1.1.2 Useful by default	112
21.1.1.3 AllowAmbiguousTypes	112
21.1.1.4 ApplicativeDo	112
21.1.1.5 ConstrainedClassMethods	113
21.1.1.6 CPP	113
21.1.1.7 DeriveFunctor	113
21.1.1.8 ExplicitForAll	113
21.1.1.9 FlexibleContexts	113
21.1.1.10 FlexibleInstances	114
21.1.1.11 GeneralizedNewtypeDeriving	114
21.1.1.12 ImplicitParams	114
21.1.1.13 LambdaCase	114
21.1.1.14 MultiParamTypeClasses	114
21.1.1.15 MultiWayIf	115
21.1.1.16 OverloadedStrings	115
21.1.1.17 PartialTypeSignatures	115
21.1.1.18 RankNTypes	115
21.1.1.19 ScopedTypeVariables	116
21.1.1.20 TupleSections	116
21.1.1.21 TypeApplications	116
21.1.1.22 TypeSynonymInstances	116
21.1.1.23 UndecidableInstances	117
21.1.1.24 ViewPatterns	117
21.1.1.25 DatatypeContexts	117
21.1.1.26 StandaloneKindSignatures	117
21.1.1.27 PartialTypeSignatures	118



21.1.2	How to make a GHC LANGUAGE extension . . . . .	118
<b>22</b>	<b>Predicative</b>	<b>120</b>
<b>23</b>	<b>Compositionality</b>	<b>121</b>
23.1	* . . . . .	121
<b>24</b>	<b><math>\Psi</math>-combinator</b>	<b>122</b>
24.1	* . . . . .	122
<b>25</b>	<b>Quantifier</b>	<b>123</b>
25.1	* . . . . .	123
25.2	Forall quantifier . . . . .	123
25.2.1	* . . . . .	123
<b>26</b>	<b>Referential transparency</b>	<b>124</b>
26.1	* . . . . .	124
<b>27</b>	<b>Relation</b>	<b>125</b>
27.1	* . . . . .	125
<b>28</b>	<b>REPL</b>	<b>126</b>
<b>29</b>	<b>Semantics</b>	<b>127</b>
29.1	Operational semantics . . . . .	127
29.2	Denotational semantics . . . . .	127
29.2.1	Abstraction . . . . .	127
29.2.1.1	* . . . . .	128
29.2.1.2	Leaky abstraction . . . . .	128
29.2.1.3	Object . . . . .	128
29.2.2	Ambigram . . . . .	129
29.2.3	Binary . . . . .	129
29.2.4	Arbitrary . . . . .	129
29.2.5	Refutable . . . . .	130
29.2.6	Irrefutable . . . . .	130
29.3	Axiomatic semantics . . . . .	130
29.3.1	Property . . . . .	130
29.3.1.1	* . . . . .	130
29.3.1.2	Associativity . . . . .	130
29.3.1.3	Left associative . . . . .	131
29.3.1.4	Right associative . . . . .	131
29.3.1.5	Non-associative . . . . .	131
29.3.1.6	Basis . . . . .	131
29.3.1.7	Commutativity . . . . .	132
29.3.1.8	Idempotence . . . . .	132
29.3.1.9	Distributive property . . . . .	132
29.4	Argument . . . . .	133
29.4.1	Argument of a function . . . . .	133
29.4.1.1	* . . . . .	133
29.5	Content word . . . . .	133
29.6	Ancient Greek and Latin prefixes . . . . .	133
29.6.1	* . . . . .	133
29.7	Idiom . . . . .	133
29.7.1	* . . . . .	133
29.8	Impredicative . . . . .	133
29.9	Context-free grammar . . . . .	135

29.9.1 *	135
<b>30 Set</b>	<b>136</b>
30.1 *	136
30.2 Closed set	136
30.3 Power set	136
30.4 Singleton	136
30.5 Russell's paradox	136
30.6 Cartesian product	137
30.6.1 Pullback	137
30.6.1.1 *	137
<b>31 Shrinking</b>	<b>138</b>
<b>32 Spine</b>	<b>139</b>
<b>33 Superclass</b>	<b>140</b>
<b>34 Tensor</b>	<b>141</b>
34.1 *	141
<b>35 Testing</b>	<b>142</b>
35.1 Property testing	142
35.1.1 Function property	142
35.1.2 Property testing type	142
35.1.3 Generator	142
35.1.3.1 *	143
35.1.3.2 Custom generator	143
35.1.4 Reusing test code	143
35.1.4.1 Test Commutative property	143
35.1.4.2 Test Symmetry property	143
35.1.4.3 Test Equivalence property	144
35.1.4.4 Test Inverse property	144
35.1.5 QuickCheck	144
35.1.5.1 Manual automation with QuickCheck properties	144
35.2 Write tests algorithm	145
<b>36 Uncurry</b>	<b>146</b>
<b>37 Unit</b>	<b>147</b>
<b>38 Nullary</b>	<b>148</b>
<b>39 Logic</b>	<b>149</b>
39.1 Proposition	149
39.1.1 *	149
39.1.2 Atomic proposition	149
39.1.2.1 *	149
39.1.3 Compound proposition	149
39.1.3.1 *	149
39.1.4 Propositional logic	149
39.1.4.1 *	150
39.1.4.2 First-order logic	150
39.2 Logical connective	150
39.2.1 *	150
39.2.2 Conjunction	150
39.2.3 Disjunction	151

39.3 Predicate	151
39.4 Statement	151
39.4.1 *	151
39.5 Iff	152
<b>40 Haskell structures</b>	<b>153</b>
40.1 Pattern match	153
40.1.1 As-pattern	153
40.1.1.1 *	153
40.1.2 Wild-card	153
40.1.2.1 *	153
40.1.3 Case	154
40.1.4 Guard	154
40.1.4.1 *	154
40.1.5 Pattern guard	154
40.1.5.1 *	155
40.1.6 Lazy pattern	155
40.1.6.1 *	155
40.1.7 Pattern binding	155
40.1.7.1 *	155
40.2 Smart constructor	155
40.3 Level of code	156
40.3.1 *	156
40.3.2 Type level	156
40.3.2.1 Type level declaration	156
40.3.2.2 Type check	156
40.3.3 Term level	157
40.3.4 Compile level	157
40.3.4.1 *	157
40.3.5 Runtime level	157
40.3.6 Kind level	157
40.3.6.1 Kind check	157
40.4 Orphan type instance	158
40.5 Undefined	158
40.6 Hierarchical module name	158
40.6.1 *	163
40.7 import	163
40.8 Let	164
40.8.1 *	164
40.9 Where	164
40.9.1 *	165
40.10 Reserved word	165
40.10.1 *	165
40.11 Haskell Language Report	165
40.11.1 *	165
40.12 Haskell'	165
40.12.1 *	165
<b>41 Computer science</b>	<b>166</b>
41.1 Guerrilla patch	166
41.1.1 Monkey patch	166
41.2 Interface	166
41.3 Module	166
41.4 Scope	166
41.4.1 Dynamic scope	166

41.4.2 Lexical scope . . . . .	166
41.4.2.1 * . . . . .	167
41.4.3 Local scope . . . . .	167
41.4.3.1 * . . . . .	167
41.5 Shadowing . . . . .	167
41.6 Syntatic sugar . . . . .	167
41.7 System F . . . . .	167
41.7.1 * . . . . .	167
41.8 Tail call . . . . .	167
41.9 Thunk . . . . .	167
41.10 Application memory . . . . .	168
41.11 Turing machine . . . . .	168
41.11.1 Turing complete . . . . .	168
41.11.1.1 * . . . . .	168
<b>42 Graph theory</b>	<b>169</b>
42.1 Successor . . . . .	169
42.1.1 Direct successor . . . . .	169
42.2 Predecessor . . . . .	169
42.2.1 Direct predecessor . . . . .	169
42.3 Degree . . . . .	169
42.3.1 Indegree . . . . .	169
42.3.2 Outdegree . . . . .	169
42.4 Adjacency matrix . . . . .	169
42.4.0.1 InstanceSigs . . . . .	170
42.5 Strongly connected . . . . .	170
42.5.1 * . . . . .	170
42.5.2 Strongly connected component . . . . .	170
42.5.2.1 * . . . . .	170
<b>43 Linear</b>	<b>171</b>
43.1 * . . . . .	171
<b>44 Stream</b>	<b>172</b>
<b>45 Bisimulation</b>	<b>173</b>
45.1 * . . . . .	173
<b>46 Syntax tree</b>	<b>174</b>
46.1 Abstract syntax tree . . . . .	174
46.1.1 * . . . . .	174
46.2 Concrete syntax tree . . . . .	174
46.2.1 * . . . . .	174
<b>47 Domain specific language</b>	<b>175</b>
47.1 * . . . . .	175
47.2 Embedded domain specific language . . . . .	175
47.2.1 * . . . . .	175
<b>48 Tagless-final</b>	<b>176</b>
<b>III Give definitions</b>	<b>177</b>
<b>49 Identity type</b>	<b>178</b>
<b>50 Constant type</b>	<b>179</b>

<b>51 Gen</b>	<b>180</b>
<b>52 Tensorial strength</b>	<b>181</b>
<b>53 Strong monad</b>	<b>182</b>
<b>54 Weak head normal form</b>	<b>183</b>
54.1 * . . . . .	183
<b>55 Function image</b>	<b>184</b>
55.1 * . . . . .	184
<b>56 Invertible</b>	<b>185</b>
<b>57 Invertibility</b>	<b>186</b>
<b>58 Define LANGUAGE pragma options</b>	<b>187</b>
58.1 ExistentialQuantification . . . . .	187
58.2 GADTs . . . . .	187
58.3 * . . . . .	187
58.4 GeneralizedNewTypeClasses . . . . .	187
58.5 FuncitonalDependencies . . . . .	187
<b>59 GHC check keys</b>	<b>188</b>
59.1 -Wno-partial-type-signatures . . . . .	188
<b>60 Generalised algebraic data types</b>	<b>189</b>
60.1 * . . . . .	189
<b>61 Order theory</b>	<b>190</b>
61.1 Domain theory . . . . .	190
61.2 Lattice . . . . .	190
61.3 Order . . . . .	190
61.3.1 Preorder . . . . .	190
61.3.1.1 * . . . . .	190
61.3.1.2 Total preorder . . . . .	190
61.3.2 Partial order . . . . .	191
61.3.2.1 * . . . . .	191
61.4 Partial order . . . . .	191
61.5 Total order . . . . .	191
<b>62 Universal algebra</b>	<b>192</b>
<b>63 Relation</b>	<b>193</b>
63.1 Reflexivity . . . . .	193
63.1.1 * . . . . .	193
63.2 Irreflexivity . . . . .	193
63.2.1 * . . . . .	193
63.3 Transitivity . . . . .	193
63.3.1 * . . . . .	194
63.4 Symmetry . . . . .	194
63.4.1 * . . . . .	194
63.5 Equivalence . . . . .	194
63.5.1 * . . . . .	194
63.6 Antisymmetry . . . . .	194
63.6.1 * . . . . .	194
63.7 Asymmetry . . . . .	194

63.7.1 *	195
<b>64 Cryptomorphism</b>	<b>196</b>
64.1 *	196
<b>65 Lexically scoped type variables</b>	<b>197</b>
<b>66 Abstract data type</b>	<b>198</b>
66.1 *	198
<b>67 Concrete type</b>	<b>199</b>
<b>68 Functional dependencies</b>	<b>200</b>
<b>69 MonoLocalBinds</b>	<b>201</b>
<b>70 KindSignatures</b>	<b>202</b>
<b>71 ExplicitNamespaces</b>	<b>203</b>
<b>72 Combinator pattern</b>	<b>204</b>
<b>73 Symbolic expression</b>	<b>205</b>
73.1 *	205
<b>74 Polynomial</b>	<b>206</b>
74.1 *	206
<b>75 Data family</b>	<b>207</b>
<b>76 Type synonym family</b>	<b>208</b>
<b>77 Indexed type family</b>	<b>209</b>
77.1 *	209
<b>78 TypeFamilies</b>	<b>210</b>
<b>79 Error</b>	<b>211</b>
79.1 *	211
<b>80 Exception</b>	<b>212</b>
80.1 *	212
<b>81 ConstraintKinds</b>	<b>213</b>
<b>82 Specialisation</b>	<b>214</b>
82.1 *	214
<b>83 Diagram</b>	<b>215</b>
<b>84 Cathegory theoretical presheaf</b>	<b>216</b>
<b>85 Topological presheaf</b>	<b>217</b>
<b>86 Diagonal functor</b>	<b>218</b>
<b>87 Limit functor</b>	<b>219</b>
<b>88 Dual vector space</b>	<b>220</b>

89 Fundamental group	221
90 Algebra of continuous function	222
91 Tangent and cotangent bundle	223
92 Group action / representation	224
93 Lie algebra	225
94 Tensor product	226
95 Forgetful functor	227
96 Free functor	228
97 Homomorphism group	229
98 Representable functor	230
99 Corecursion	231
100 Coinduction	232
101 Initial algebra of an endofunctor	233
102 Terminal coalgebra for an endofunctor	234
IV Citations	235
V Good code	237
103 Good: Type aliasing	238
104 Good: Type wideness	239
105 Good: Print	240
106 Good: Fold	241
107 Good: Computation model	242
108 Good: Make bottoms only local	243
109 Good: Newtype wrap is ideally transparent for compiler and does not change performance	244
110 Good: Instances of types/type classes must go with code you write	245
111 Good: Functions can be abstracted as arguments	246
112 Good: Infix operators can be bind to arguments	247
113 Good: Arbitrary	248
114 Good: Principle of Separation of concerns	249
115 Good: Function composition	250

<b>116Good: Point-free</b>	<b>251</b>
116.1Good: Point-free is great in multi-dimention	251
<b>117Good: Functor application</b>	<b>252</b>
<b>118Good: Parameter order</b>	<b>253</b>
<b>119Good: Applicative monoid</b>	<b>254</b>
<b>120Good: Creative process</b>	<b>255</b>
120.1Pick phylosophy principles one to three the more - the harder the implementation	255
120.2Draw the most blurred representation	255
120.3Deduce abstractions and write remotely what they are	255
120.4Model of computation	255
120.4.1 Model the domain	255
120.4.2 Model the types	255
120.4.3 Think how to write computations	255
120.5Create	255
<b>121«Good: About operators (&lt;\$ ) ( **&gt;) (&lt;* ) (&gt; ) »&gt;</b>	<b>256</b>
<b>122Good: About functions like {mapM, sequence}_</b>	<b>257</b>
<b>123Good: Guideliles</b>	<b>258</b>
123.1Wiki.haskell	258
123.1.1 Documentation	258
123.1.1.1 Comments write in application terms, not technical.	258
123.1.1.2 Tell what code needs to do not how it does.	258
123.1.2 Haddock	258
123.1.2.1 Put haddock comments to ever exposed data type and function.	258
123.1.2.2 Haddock header	258
123.1.3 Code	258
123.1.3.1 Try to stay closer to portable (Haskell98) code	258
123.1.3.2 Try make lines no longer 80 chars	258
123.1.3.3 Last char in file should be newline	258
123.1.3.4 Symbolic infix identifiers is only library writer right	258
123.1.3.5 Every function does one thing.	258
<b>124Good: Use Typed holes to progress the code</b>	<b>259</b>
<b>125Good: Haskell allows infinite terms but not infinite types</b>	<b>260</b>
<b>126Good: Use type sysonims to differ the information</b>	<b>261</b>
<b>127«Good: Control.Monad.Error -&gt; Control.Monad.Except»&gt;</b>	<b>262</b>
<b>128Good: Monad OR Applicative</b>	<b>263</b>
128.0.1Start writing monad using 'return', 'ap', 'liftM', 'liftM2', '»' instead of 'do', '»='	263
128.0.2Basic case when Applicative can be used	263
128.0.3Applicative block vs Monad block	263
<b>129Good: Haskell Package Versioning Policy</b>	<b>264</b>
129.1 *	264
<b>130Good: Linear type</b>	<b>265</b>
<b>131Good: Exception vs Error</b>	<b>266</b>



132	Good: Let vs. Where	267
133	Good: RankNTypes	268
134	Good: Orphan type instance	269
135	Good: Smart constructor	270
136	Good: Thin category	271
137	Good: Recursion	272
138	Good: Monoid	273
139	Good: Free monad	274
140	Good: Use mostly where clauses	275
141	Good: Where clause is in a scope with function parameters	276
142	Good: Strong preference towards pattern matching over {head, tail, etc.} functions	277
143	Good: Patternmatching is possible on monadic bind in do	278
144	Good: Applicative vs Monad	279
145	Good: StateT, ReaderT, WriterT	280
146	Good: Working with MonadTrans and lift	281
147	Good: Don't mix Where and Let	282
148	Good: Where vs. Let	283
149	Good: The proper nature algorithm that models behaviour of many objects is computation heavy	284
150	Good: In Haskell parameters bound by lambda declaration instantiate to only one concrete type	285
151	Good: Instance is a good structure to draw a type line	286
152	Good: MTL vs. Transformers	287
VI	Bad code	288
153	Bad pragma	289
153.1	Bad: Dangerous LANGUAGE pragma option . . . . .	289
VII	Useful functions to remember	290
154	Prelude	291
154.1	Ord . . . . .	291
154.2	Calc . . . . .	291
154.3	List operations . . . . .	291

<b>155</b>	<b>Data.List</b>	<b>292</b>
<b>156</b>	<b>Data.Char</b>	<b>293</b>
<b>157</b>	<b>QuickCheck</b>	<b>294</b>
<b>VIII</b>	<b>Tools</b>	<b>295</b>
<b>158</b>	<b>ghc-pkg</b>	<b>296</b>
<b>159</b>	<b>Search over the Haskell packages code: Codesearch from Aelve</b>	<b>297</b>
<b>160</b>	<b>Integration of NixOS/Nix with Haskell IDE Engine (HIE) and Emacs (Spacemacs)</b>	<b>298</b>
160.11.	Install the Cachix: <a href="https://github.com/cachix/cachix">https://github.com/cachix/cachix</a>	298
160.22.	Installation of HIE: <a href="https://github.com/infinisil/all-hies/#cached-builds">https://github.com/infinisil/all-hies/#cached-builds</a>	298
160.2.1 2.1.	Provide cached builds	298
160.2.2 2.2.a.	Installation on NixOS distribution:	298
160.2.3 2.2.b.	Installation with Nix package manager:	299
160.33.	Emacs (Spacemacs) configuration:	299
160.44.	Open the Haskell file from a project	300
160.55.	Be pleased writing code	300
160.66.	(optional) Debugging	300
<b>161</b>	<b>Debugger</b>	<b>302</b>
<b>162</b>	<b>GHCI</b>	<b>303</b>
<b>IX</b>	<b>Libs</b>	<b>304</b>
<b>163</b>	<b>Exceptions</b>	<b>305</b>
163.1	Exceptions - optionally pure extensible exceptions that are compatible with the mtl	305
163.2	Safe-exceptions - safe, simple API equivalent to the underlying implementation in terms of power, encourages best practices minimizing the chances of getting the exception handling wrong.	305
163.3	Enclosed-exceptions - capture exceptions from the enclosed computation, while reacting to asynchronous exceptions aimed at the calling thread.	305
<b>164</b>	<b>Memory management</b>	<b>306</b>
164.1	membrain - type-safe memory units	306
<b>165</b>	<b>Parsers - megaparsec</b>	<b>307</b>
<b>166</b>	<b>CLIs - optparse-applicative</b>	<b>308</b>
<b>167</b>	<b>HTML - Lucid</b>	<b>309</b>
<b>168</b>	<b>Web applications - Servant</b>	<b>310</b>
<b>169</b>	<b>IO libraries</b>	<b>311</b>
169.1	Conduit - practical, monolythic, guarantees termination return	311
169.2	Pipes + Pipes Parse - modular, more primitive, theoretically driven	311
<b>170</b>	<b>JSON - aeson</b>	<b>312</b>
<b>171</b>	<b>Backpack</b>	<b>313</b>

<b>X</b>	<b>Drafts</b>	<b>314</b>
<b>172</b>	<b>Exception handling</b>	<b>315</b>
<b>173</b>	<b>Constraints</b>	<b>318</b>
<b>174</b>	<b>Monad transformers and their type classes</b>	<b>320</b>
<b>175</b>	<b>Layering monad transformers</b>	<b>321</b>
<b>176</b>	<b>Hoogle</b>	<b>322</b>
176.1	Search . . . . .	322
176.2	Scope . . . . .	322
176.2.1	Default . . . . .	322
176.2.2	Hierarchical module name system (from big letter): . . . . .	322
176.2.3	Packages (lower case): . . . . .	323
<b>177</b>	<b>ST-Trick monad</b>	<b>324</b>
177.1	* . . . . .	324
<b>178</b>	<b>Either</b>	<b>325</b>
178.1	* . . . . .	325
<b>179</b>	<b>Inverse</b>	<b>326</b>
<b>180</b>	<b>Inversion</b>	<b>327</b>
<b>181</b>	<b>Inverse function</b>	<b>328</b>
<b>182</b>	<b>Inverse morphism</b>	<b>329</b>
<b>183</b>	<b>Partial inverse</b>	<b>330</b>
<b>184</b>	<b>PatternSynonyms</b>	<b>331</b>
184.1	* . . . . .	331
<b>185</b>	<b>GHC debug keys</b>	<b>332</b>
185.1	ddump-ds . . . . .	332
185.1.1	* . . . . .	332
<b>186</b>	<b>GHC optimize keys</b>	<b>333</b>
186.1	foptimal-applicative-do . . . . .	333
<b>187</b>	<b>Computational trinitarianism</b>	<b>334</b>
187.1	* . . . . .	335
<b>188</b>	<b>Techniques functional programming deals with the state</b>	<b>336</b>
188.1	Minimizing . . . . .	336
188.2	Concentrating . . . . .	336
188.3	Deferring . . . . .	336
<b>189</b>	<b>Monadic Error handling</b>	<b>337</b>
<b>190</b>	<b>Functions</b>	<b>338</b>
<b>191</b>	<b>Void</b>	<b>339</b>
191.1	* . . . . .	339
<b>192</b>	<b>Constructive proof</b>	<b>340</b>

<b>193</b>	<b>Intuitionistic logic</b>	<b>341</b>
193.1 *	.....	341
<b>194</b>	<b>Principle of explosion</b>	<b>342</b>
194.1 *	.....	342
<b>195</b>	<b>Universal property</b>	<b>343</b>
<b>196</b>	<b>Yoneda lemma</b>	<b>344</b>
<b>197</b>	<b>Monoidal category, functoriality of ADTs, Profunctors</b>	<b>345</b>
<b>198</b>	<b>Const functor</b>	<b>346</b>
<b>199</b>	<b>Arrow in Haskell</b>	<b>347</b>
<b>200</b>	<b>Contravariant functor</b>	<b>348</b>
<b>201</b>	<b>Profunctor</b>	<b>349</b>
<b>XI</b>	<b>Reference</b>	<b>350</b>
<b>202</b>	<b>Functor-Applicative-Monad Proposal</b>	<b>351</b>
202.1 *	.....	351
<b>203</b>	<b>Haskell-98</b>	<b>352</b>
203.1	Old instance termination rules .....	352
<b>204</b>	<b>Performance results and comparisons of types &amp; solutions</b>	<b>353</b>
<b>XII</b>	<b>Liturgy</b>	<b>354</b>

# Contents

# Part I

## Introduction

---

*“Employ your time in improving yourself by other men’s writings so that you shall come easily by what others have labored hard for.”*  
(Socrates by Plato)

Important notes on Haskell, [category](#) theory & related fields, terms and recommendations.

Resources:

- Web book: <https://blog.latukha.com/haskell-notes>
- GitHub: <https://github.com/Anton-Latukha/haskell-notes>
- GitLab: <https://gitlab.com/Anton.Latukha/haskell-notes>
- View PDF: <https://github.com/Anton-Latukha/haskell-notes/blob/master/README.pdf>
- Download PDF: <https://github.com/Anton-Latukha/haskell-notes/raw/master/README.pdf>
- L<sup>A</sup>T<sub>E</sub>X: <https://github.com/Anton-Latukha/haskell-notes/raw/master/README.tex>

This is a complex Org markup file with L<sup>A</sup>T<sub>E</sub>X formulas.  
GitHub & GitLab only partially parse Org into HTML.

To get the full view:

- Outline navigation
- L<sup>A</sup>T<sub>E</sub>X formulas: 
$$\left[-\frac{\hbar^2}{2m}\nabla^2 + V(\vec{r}, t)\right]\Psi(\vec{r}, t) = i\hbar\frac{\partial}{\partial t}\Psi(\vec{r}, t), \quad \sum_{k,j}\left[-\frac{\hbar^2}{\sqrt{a}}\frac{\partial}{\partial q^k}\left(\sqrt{a}a^{kj}\frac{\partial}{\partial q^j}\right) + V\right]\Psi + \frac{\hbar}{i}\frac{\partial\Psi}{\partial t} = 0$$
- [Interlinks](#): Interlinks

, please refer to Org mode capable viewer/editor, or to the web book.

Note about markup: «<This is a radio target>>» - for dynamic org-mode linking.

To prettify radio targets in Emacs with Elisp snippet to prettify «<Radio targets>>» to Radio targets:

```
;;; 2019-06-12: NOTE: Prettify '«<<Radio targets>>>»' to be shown as '_Radio_targets_' when 'org-
;;; This is improvement of the code from: Tobias&glmorous: https://emacs.stackexchange.com/quest
;;; There exists library created from the sample: https://github.com/talwrii/org-hide-targets
(defcustom org-hidden-links-additional-re "\\(«<<\\)\\[[:print:]]+?\\(>>>\\)"
  "Regular expression that matches strings where the invisible-property of the sub-matches 1 and
:type '(choice (const :tag "Off" nil) regexp)
:group 'org-link)
```

---

```
(make-variable-buffer-local 'org-hidden-links-additional-re)

(defun org-activate-hidden-links-additional (limit)
  "Put invisible-property org-link on strings matching 'org-hide-links-additional-re'."
  (if org-hidden-links-additional-re
      (re-search-forward org-hidden-links-additional-re limit t)
      (goto-char limit)
      nil))

(defun org-hidden-links-hook-function ()
  "Add rule for 'org-activate-hidden-links-additional' to 'org-font-lock-extra-keywords'.
  You can include this function in 'org-font-lock-set-keywords-hook'."
  (add-to-list 'org-font-lock-extra-keywords
    '(org-activate-hidden-links-additional
      (1 '(face org-target invisible org-link))
      (2 '(face org-target invisible org-link)))))

(add-hook 'org-font-lock-set-keywords-hook #'org-hidden-links-hook-function)

SCHT: and metadata in :properties: - of my org-drill practices, please just run org-drill-strip-all-data.
```



# Part II

## Definitions

# Chapter 1

## Algebra

al-jabr - assemble parts.

A system of parts based on given axioms ([properties](#)).

—

- a. [Abstract algebra](#) - the study of number systems and operations within them.
- b. [Algebra](#) - vector space over a field with a multiplication.
- c. [Algebra](#) - a [set](#) with its [algebraic structure](#).

### 1.1 \*

Algebras

### 1.2 Algebraic

Composite from simple parts.

Also: [Algebraic data type](#).

### 1.3 Algebraic structure

[Algebraic structure](#) on a [set](#) (called carrier [set](#) or underlying [set](#)) is a collection of finitary operations on that [set](#).

The [set](#) with this [structure](#) is also called an [algebra](#).

[Algebraic structures](#) include [groups](#), [rings](#), fields, and lattices. More complex [structures](#) can be defined by introducing multiple operations, different underlying [sets](#), or by altering the defining axioms. Examples of more complex [algebraic structures](#) include vector spaces, modules, and [algebras](#).

Table 1.1: Algebraic structures

	Closure	Associativity	Identity	Invertability	Commutativity
Semigroupoid		✓			
Small Category		✓	✓		
Groupoid		✓	✓	✓	
Magma	✓				
Quasigroup	✓			✓	
Loop	✓		✓	✓	
Semigroup	✓	✓			
Inverse Semigroup	✓	✓		✓	
Monoid	✓	✓	✓		
Group	✓	✓	✓	✓	
Abelian group	✓	✓	✓	✓	✓
Ring	✓	✓	✓	✓	under +

### 1.3.1 \*

Algebraic structures

### 1.3.2 Fundamental theorem of algebra

Any non-constant single-variable polynomial with complex coefficients has at least one complex root.

From this definition follows property that the field of complex numbers is algebraically closed.

## 1.4 Modular arithmetic

System for integers where numbers wrap around the certain values (single - modulus, plural - moduli).

Example - 12-hour clock.

### 1.4.1 \*

Clock arithmetic

### 1.4.2 Modulus

Special numbers where arithmetic wraps around in modular arithmetic.

#### 1.4.2.1 \*

Moduli - plural.

# Chapter 2

## Bind

Establishing equality between two [objects](#).

Most often:

- equating [variable](#) to a value.
- equating [parameter](#) of a [function](#) to an [argument](#) ([variable](#)/value/[function](#)). This term often is equated to [applying argument](#) to a [function](#), which includes  [\$\beta\$ -reduction](#).

### 2.1 \*

Binds  
Binding  
Bindings

## Chapter 3

# Category theory

Category  $\mathcal{C}$  consists of the **basis**:

Primitives:

- a. **Objects** -  $a^{\mathcal{C}}$ . A **node**. **Object** of some **type**. Often **sets**, than it is **Set category**.
- b. **Arrows** -  $(a, b)^{\mathcal{C}}$  (AKA **morphisms** mappings).
- c. **Arrow (morphism) composition** - **binary operation**:  $(a, b)^{\mathcal{C}} \circ (b, c)^{\mathcal{C}} \equiv (a, c)^{\mathcal{C}} \mid \forall a, b, c \in \mathcal{C}$ .  
AKA principle of **compositionality** for **arrows**.

**Properties** (or axioms):

- a. **Associativity** of **morphisms**:  $h \circ (g \circ f) \equiv (h \circ g) \circ f \mid f_{a \rightarrow b}, g_{b \rightarrow c}, h_{c \rightarrow d}$ .
- b. Every **object** has (two-sided) **identity morphism** ( & in fact - exactly one):  $1_x \circ f_{a \rightarrow x} \equiv f_{a \rightarrow x}, g_{x \rightarrow b} \circ 1_x \equiv g_{x \rightarrow b} \mid \forall x \exists 1_x, \forall f_{a \rightarrow x}, \forall g_{x \rightarrow b}$ .
- c. Principle of **compositionality**.

From these axioms, can be proven that there is exactly one **identity morphism** for every **object**.

**Object** and **morphism** are complete **abstractions** for anything.

In majority of cases under **object** is a state and **morphism** is a change.

### 3.1 \*

Category  
Categories

## 3.2 Abelian category

Generalised [category](#) for homological [algebra](#) (having a possibility of basic constructions and techniques for it).

[Category](#) which:

- has a [zero object](#),
- has all [binary](#) biproducts,
- has all [kernel](#)'s and cokernels,
- (it has all [pullbacks](#) and pushouts)
- all [monomorphism](#)'s and [epimorphism](#)'s are normal.

[Abelian category](#) is very stable; for example they are regular and they satisfy the snake lemma. The class of [Abelian categories](#) is [closed](#) under several categorical constructions.

There is notion of [Abelian monoid](#) (AKS [Commutative monoid](#)) and [Abelian group](#) ([Commutative group](#)).

Basic examples of  $*$ :

- [category](#) of Abelian [groups](#)
- [category](#) of modules over a [ring](#).

$*$  are widely used in [algebra](#), [algebraic](#) geometry, and topology.

$*$  has many constructions like in [categories](#) of modules:

- kernels
- exact [sequences](#)
- [commutative](#) diagrams

$*$  has disadvantage over [category](#) of modules. [Objects](#) do not necessarily have elements that can be manipulated directly, so traditional definitions do not work. Methods must be supplied that allow definition and manipulation of [objects](#) without the use of elements.

### 3.2.1 $*$

Abelian categories

### 3.3 Composition

Axiom of [Category](#).

#### 3.3.1 \*

Composable  
Compositions

### 3.4 Endofunctor category

From the name, in this [Category](#):

- [objects](#) of *End* are [Endofunctors](#)  $E^{C \rightarrow C}$
- [morphisms](#) are [natural transformations](#) between [endofunctors](#)

### 3.5 Functor

\* full translation (map) of one [category](#) into another.

Translating [objects](#) and [morphisms](#) (as input can take [morphism](#) or [object](#)).

\* - [forgetful](#) - discards part of the [structure](#).

\* - faithful - fully preserves all [morphisms](#) - [injective](#) on Hom-sets.

\* - full - translation of [morphisms](#) fully covers all the [morphisms](#) between according objects in the target category.

For [Functor type class](#) or [fmap](#) - see [Power set functor](#).

[Functor properties](#) (axioms):

- $F^{C \rightarrow D}(a) \mid \forall a^C$  - every source [object](#) is mapped to [object](#) in target [category](#)
- $\overrightarrow{(F^{C \rightarrow D}(a), F^{C \rightarrow D}(b))}^D \mid \forall \overrightarrow{(a, b)}^C$  - every source [morphism](#) is mapped to target [category morphism](#) between corresponding [objects](#)
- $F^{C \rightarrow D}(\overrightarrow{g}^C \circ \overrightarrow{f}^C) = F^{C \rightarrow D}(\overrightarrow{g}^C) \circ F^{C \rightarrow D}(\overrightarrow{f}^C) \mid \forall y = \overrightarrow{f}^C(x), \forall \overrightarrow{g}^C(y)$  - [composition](#) of [morphisms](#) translates directly (tautologically goes from other two)

These axioms guarantee that [composition](#) of [functors](#) can be fused into one [functor](#) with [composition](#) of [morphisms](#). This process called [fusion](#).

In Haskell this axioms have form:

```
fmap id = id
fmap (f . g) = fmap f . fmap g
```

Since `*` is 1-1 mapping of initial [objects](#) - it is a memoizable dictionary with [cardinality](#) of initial [objects](#). Also in [Hask category functors](#) are obviously [endofunctors](#)  $\therefore$  they are special [kinds](#) of containers for the parametric values (AKA [product type](#)). In Haskell [product type](#) `*` are [endofunctors](#) from [polymorphic type](#) into a [functor](#) wrapper of a [polymorphic type](#).

`*` translates in one direction, and does not provide algorithm of reversing itself or retrieving the parametric value.

### 3.5.1 `*`

Functors

### 3.5.2 Power set functor

$\mathcal{P}^S \rightarrow \mathcal{P}(S)$

`*` - [functor](#) from [set](#)  $S$  to its [power set](#)  $\mathcal{P}(S)$ .

[Functor type class](#) in Haskell defines a `*` and allows to do [function application](#) inside [type structure](#) layers (denoted  $f$  or  $m$ ). [IO](#) is also such [structure](#).

[Power set](#) is unique to the [set](#), `*` is unique to the [category](#) ([data type](#)).

`*` embodies in itself any [endofunctor](#). It is easily seen from Haskell definition - that the `*` is the [polymorphic](#) generalization over any [endofunctor](#) in a [category](#). [Application](#) of a [function](#) to `*` gives a particular [endofunctor](#) (see [Hask category](#)).

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

[Functor](#) instance must be of [kind](#)  $( * \rightarrow * )$ , so instance for [higher-kinded data type](#) must be [applied](#) until this [kind](#).

[Composed](#) `*` can [lift functions](#) through any layers of [structures](#) that belong to [Functor type class](#).

`*` can be used to filter-out [error](#) cases ([Nothing](#) & Left cases) in [Maybe](#), [Either](#) and related [types](#).

#### 3.5.2.1 `*`

fmap  
Functor type class

#### 3.5.2.2 Power set functor laws

[Type](#) instance of [functor](#) should abide this laws:

a. `*`

Functor laws

b. Power set functor identity law

```
fmap id == id
```



c. Power set functor composition law

```
fmap (f.g) == fmap f . fmap g
```

In words, it is if several **functions** are **composed** and then **fmap** is **applied** on them - it should be the same as if **functions** was **fmapped** and then **composed**.

### 3.5.2.3 Lift

```
fmap :: (a -> b) -> (f a -> f b)
```

**Functor** takes **function**  $a \rightarrow b$  and returns a **function**  $f\ a \rightarrow f\ b$  this is called **lifting** a **function**. **Lift** does a **function application** through the **data structure**.

a. \*

Lifting

### 3.5.2.4 Power set functor is a free monad

Since:

- $\forall e \in S : \exists \{e\} \in \mathcal{P}(S) \models \forall e \in S : \exists (e \rightarrow \{e\}) \equiv unit$
- $\forall \mathcal{P}(S) : \mathcal{P}(S) \in \mathcal{P}(S) \models \forall \mathcal{P}(S) : \exists (\mathcal{P}(\mathcal{P}(S)) \rightarrow \mathcal{P}(S)) \equiv join$

### 3.5.3 Functorial

Corresponds to **functor laws**.

### 3.5.4 Forgetful functor

**Functor** that forgets part or all of what defines **structure** in **domain category**.

$F^{Grp \rightarrow Set}$  that translates **groups** into their underlying **sets**.

**Constant functor** is another example.

#### 3.5.4.1 \*

Forgetful

### 3.5.5 Identity functor

Maps all **category** to itself. All **objects** and **morphisms** to themselves.

Denotation:

$1^{C \rightarrow C}$

### 3.5.6 Endofunctor

Is a [functor](#) which source ([domain](#)) and target ([codomain](#)) are the same [category](#).

$$F^{C \rightarrow C}, E^{C \rightarrow C}$$

#### 3.5.6.1 \*

Endofunctors

### 3.5.7 Applicative functor

\* - Computer science term. [Category](#) theory name - [lax monoidal functor](#). And in [category](#) *Set*, and so in [category](#) *Hask* all [applicatives](#) and [monads](#) are strong (have [tensorial strength](#)).

\* - [sequences functorial](#) computations (plain [functors](#) can't).

```
(<*>) :: f (a -> b) -> f a -> f b
```

Requires [Functor](#) to exist.

Requires [Monoidal structure](#).

Has [monoidal structure](#) rules, separated from [function application](#) inside [structure](#).

[Data type](#) can have several [applicative](#) implementations.

Standard definition:

```
class Functor f => Applicative f
  where
    (<*>) :: f (a -> b) -> f a -> f b
    pure  :: a -> f a
```

`pure` - if a [functor](#), [identity Kleisli arrow](#), [natural transformation](#).

[Composition](#) of `*` always produces `*`, contrary to [monad](#) ([monads](#) are not [closed](#) under [composition](#)).

`Control.Monad` has an old [function](#) `ap` that is old implementation of `<*>`:

```
ap :: Monad m => m (a -> b) -> m a -> m b
```

#### 3.5.7.1 \*

[Applicative](#)

[Applicatives](#)

[Applicative functors](#)

**3.5.7.2 Applicative law****3.5.7.3 \***

Applicative laws

- a. Applicative identity law

```
pure id <*> v = v
```

- b. Applicative composition law [Function composition](#) works regularly.

```
pure (.) <*> u <*> v <*> w = u <*> (v <*> w)
```

- c. Applicative homomorphism law Internal [function application](#) doesn't change the [structure](#) around values.

```
pure f <*> pure x = pure (f x)
```

- d. Applicative interchange law On condition that internal [order](#) of [evaluation](#) is preserved - [order](#) of operands is not relevant.

```
u <*> pure y = pure ($ y) <*> u
```

**3.5.7.4 Applicative function**

- a. liftA\*

- a. liftA Essentially a [fmap](#).

```
:type liftA
liftA :: Applicative f => (a -> b) -> f a -> f b
Lifts function into applicative function.
```

- b. liftA2 Lifts [binary function](#) across two [Applicative functors](#).

```
liftA2 :: Applicative f => (a -> b -> c) -> f a -> f b -> f c
liftA2 f x y == pure f <*> x <*> y
```

- c. «[liftA2 \(<\\*>\)](#)» [liftA2 \(<\\*>\)](#) is pretty useful. It can [lift binary operation](#) through the two layers:  
It is two-layer [Applicative](#).

```
liftA2 :: Applicative f => (    a        -> b -> c ) -> f    a        -> f    b    -
<*> :: Applicative f =>    (f  (a -> b) -> f a -> f b)
liftA2 (<*>) :: (Applicative f1, Applicative f2) =>    f1 (f2 (a -> b)) -> f1 (f2 a) -
```

- d. «[liftA2 \(liftA2 \(<\\*>\)\)](#)» [liftA2 \(<\\*>\)](#) 3-layer version.

- e. liftA3 [liftA2 3-parameter](#) version.

```
liftA3 f x y z == pure f <*> x <*> y <*> z
```

- b. Conditional [applicative](#) computations

```
when :: Applicative f => Bool -> f () -> f ()
```

Only when `True` - perform an [applicative](#) computation.

```
unless :: Applicative f => Bool -> f () -> f ()
```

Only when `False` - perform an [applicative](#) computation.

### 3.5.7.5 Special applicatives

- a. Identity applicative

```
-- Applicative f =>
-- f ~ Identity
type Id = Identity
instance Applicative Id
  where
    pure :: a -> Id a
    (<*>) :: Id (a -> b) -> Id a -> Id b

mkId = Identity
xs = [1, 2, 3]

const <$> mkId xs <*> mkId xs'
-- [1,2,3]
```

- b. Constant applicative It holds only to one value. The [function](#) does not exist and last [parameter](#) is a phantom.

```
-- Applicative f =>
-- f ~ Constant e
type C = Constant
instance Applicative C
  where
    pure :: a -> C e a
    (<*>) :: C e (a -> b) -> C e a -> C e b
```

- c. Maybe applicative "There also can be no [function](#) at all."

If [function](#) might not exist - embed `f` in [Maybe structure](#), and use [Maybe applicative](#).

```
-- f ~ Maybe
type M = Maybe
pure :: a -> M a
(<*>) :: M (a -> b) -> M a -> M b
```

- d. Either applicative `pure` is `Right`.  
 Defaults to `Left`.  
 And if there is two `Left`'s - to `Left` of the first `argument`.
- e. Validation applicative The Validation `data type` isomorphic to `Either`, but has accumulative `Applicative` on the `Left` side.  
 Validation `data type` is not a `monad`. Validation is an example of, "An `applicative functor` that is not a `monad`."  
 While `Either monad` on `Left case` just drops computation and returns this first `Left`.  
`Monad` needs to process the result of computation - it requires to be able to process all `Left error statement` cases for Validation, it is or non-terminating `Monad` or one which is impossible to implement in `polymorphic` way with Validation.

### 3.5.7.6 Monad

*monos* sole  
*monáda* `unit`

\* - `monoid` in `endofunctor category` with  $\eta$  (`unit`) and  $\mu$  (`join`) `natural transformations`.

`Monad` on  $\mathcal{C}$  is  $\{E^{\mathcal{C} \rightarrow \mathcal{C}}, \eta, \mu\}$ :

- $E^{\mathcal{C} \rightarrow \mathcal{C}}$  - is an `endofunctor`
- two `natural transformations`,  $1^{\mathcal{C}} \rightarrow E$  and  $E \circ E \rightarrow E$ :
  - $\eta^{1^{\mathcal{C}} \rightarrow E} = \text{unit}^{Identity \rightarrow E}(x) = f^{x \rightarrow E(x)}(x)$
  - $\mu^{(E \circ E) \rightarrow E} = \text{join}^{(E \circ E) \rightarrow (Identity \circ E)}(x) = |y = E(x)| = f^{E(y) \rightarrow y}(y)$

where:

- $\mathcal{C}$  is a `category`
- $1^{\mathcal{C}}$  denotes the  $\mathcal{C}$  `identity functor`
- $(E \circ E)$  - `endofunctor`  $\mathcal{C} \rightarrow \mathcal{C}$

Definition with  $\{E^{\mathcal{C} \rightarrow \mathcal{C}}, \eta, \mu\}$  (in `Hask`:  $(\{e :: f a \rightarrow f b, \text{pure}, \text{join}\})$ ) - is classic categorical, in Haskell minimal complete definition is  $\{fmap, \text{pure}, (>=>)\}$ .

If there is a `structure`  $S$ , and a way of taking `object`  $x$  into  $S$  and a way of collapsing  $S \circ S$  - there probably a `monad`.

Mostly `monads` used for sequencing actions (computations) (that looks like imperative programming), with ability to depend on previous chains. Note if `monad` is `commutative` - it does not `order` actions.

`Monad` can shorten/terminate [sequence](#) of computations. It is implemented inside `Monad` instance. For example [Maybe monad](#) on [Nothing](#) drops chain of computation and returns [Nothing](#).

\* inherits the [Applicative](#) instance methods:

```
import Control.Monad (ap)
return == pure
ap == (<*>) -- + Monad requirement
```

Table 3.1: `Monad` in mathematics and Haskell

Math	Meaning	Cat/Fctr	$X \in C$	Type	Haskell
$Id$	<a href="#">endofunctor</a> "Id"	$C \rightarrow C$	$X \rightarrow Id(X)$	$a \rightarrow a$	<a href="#">id</a>
$E$	<a href="#">endofunctor</a> "monad"	$C \rightarrow C$	$X \rightarrow E(X)$	$m\ a \rightarrow m\ b$	<a href="#">fmap</a>
$\eta$	<a href="#">natural transformation</a> "unit"	$Id \rightarrow E$	$Id(X) \rightarrow E(X)$	$a \rightarrow m\ a$	<a href="#">pure</a>
$\mu$	<a href="#">natural transformation</a> "multiplication"	$E \circ E \rightarrow E$	$E(E(X)) \rightarrow E(X)$	$m\ (m\ a) \rightarrow m\ a$	<a href="#">join</a>

Internals of `Monad` are Haskell [data types](#), and as such - they can be consumed any number of times.

[Composition](#) of [monadic types](#) does not always results in [monadic type](#).

a. \*

Monads  
Monadic

b. Monad law `Monad` corresponds to [functor laws](#) & [applicative laws](#) and additionally:

a. \* Monad laws

b. Monad left identity law

```
pure x >>= f == f x
```

Explanation:

```
>>= :: Monad f => f a -> (a -> f b) -> f b
      pure x >>= f == f x
```

Rule that `>>=` must get first [argument structure](#) internals and [apply](#) to the [function](#) that is the second [argument](#).

c. Monad right identity law

```
f >>= pure == f
```

Explanation:

```
>>= :: Monad f => f a -> (a -> f b) -> f b
      f >>= pure == f
```

AKA it is a [tacit](#) description of a [monad bind](#) as [endofunctor](#).

d. Monad associativity law

```
(m >>= f) >>= g == m >>= (\ x -> f x >>= g)
```

c. Monad type class

```
class Applicative m => Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
  (>>) :: m a -> m b -> m b
  return :: a -> m a
```

a. MonadPlus type class Is a [monoid](#) over [monad](#), with additional rules.

The precise [set](#) of rules ([properties](#)) not agreed upon. Class instances obey [monoid](#) & [left zero](#) rules, some additionally obey [left catch](#) and others [left distribution](#).

Overall there \* currently reforms ([MonadPlus](#) reform proposal) in several smaller nad strictly defined [type classes](#).

Subclass of an [Alternative](#).

a. \*

Monadplus

d. [Functor](#) -> [Applicative](#) -> [Monad](#) progression

```
<$> :: Functor    f => (a -> b) -> f a -> f b
<*> :: Applicative f => f (a -> b) -> f a -> f b
=<< :: Monad      f => (a -> f b) -> f a -> f b
```

[pure](#) & [join](#) are [Natural transformations](#) for the [fmap](#).

e. Monad function

a. Return function

```
return == pure
```

Nonstrict.

b. Join function

```
join :: Monad m => m (m a) -> m a
```

Generales knowledge of [concat](#).

[Kleisli composition](#) that flattens two layers of [structure](#) into one.

The way to express ordering in [lambda calculus](#) is to nest.

a. \*

join

b. [join](#) . [fmap](#) == ([=<<](#))

```

-- b = f b
fmap      :: Monad f => (a -> f b) -> f a -> f (f b)
join      :: Monad f =>                f (f a) -> f a
join . fmap :: Monad f => (a -> f b) -> f a          -> f b
flip      >>= :: Monad f => (a -> f b) -> f a          -> f b

```

c. Bind function

```

>>=      :: Monad f => f a -> (a -> f b) -> f b
join . fmap :: Monad f => (a -> f b) -> f a -> f b
Nonstrict.

```

The most ubiquitous way to `>>=` something is to use [Lambda function](#):

```

getLine >>= \name -> putStrLn "age pls:"

```

Also very neat way is to bundle and handle [Monad](#) - is to bundle it with [bind](#), and leave [applied](#) partially.

And use that partial bundle as a [function](#) - every [evaluation](#) of the [function](#) would trigger [evaluation](#) of internal [Monad structure](#). Thumbs up.

```

printOneOf Bool IO ()
printOneOf False = putStr "1"
printOneOf True = putStr "2"

quant (Bool IO b) IO b
quant = (>>=) (randomRIO (False, True))

recursePrintOneOf Monad m (t m a) t m b
recursePrintOneOf f x = (f x) >> (recursePrintOneOf f x)

main IO ()
main = recursePrintOneOf (quant) $ printOneOf

a. *
   Monadic extend
   Monadic bind
   Monad bind
   Binder
   Binder function

a. (»=)
b. »=
c. (=<)
d. =«

```

d. Sequencing operator `(») == (*>)`: Discard any resulting value of the action and [sequence](#) next action.

[Applicative](#) has a similar [operator](#).

```

(>>) :: m a -> m b -> m b
(*>) :: f a -> f b -> f b

```

e. [Monadic](#) versions of [list functions](#)

```

sequence :: (Traversable t, Monad m) => t (m a) -> m (t a)

```



`Sequence` gets the traversable of `monadic` computations and swaps it into `monad` computation of traverse. In the result the collection of `monadic` computations turns into one long `monadic` computation on traverse of data.

If some step of this long computation fails - `monad` fails.

```
mapM :: (Traversable t, Monad m) => (a -> m b) -> t a -> m (t b)
```

`mapM` gets the `AMB function`, then takes traversable data. Then applies `AMB function` to traversable data, and returns converted `monadic` traversable data.

```
foldM :: (Foldable t, Monad m) => (b -> a -> m b) -> b -> t a -> m b
foldl :: Foldable t           => (b -> a -> b) -> b -> t a -> b
```

`*` is a `monadic foldl`.

`b` is initial cumulative value, `m b` is a cumulative bank.

Right folding achieved by reversing the input `list`.

```
filterM :: Applicative m => (a -> m Bool) -> [a] -> m [a]
filter :: (a -> Bool) -> [a] -> [a]
```

Take Boolean `monadic` computation, filter the `list` by it.

```
zipWithM :: Applicative m => (a -> b -> m c) -> [a] -> [b] -> m [c]
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
```

Take `monadic` combine `function` and combine two lists with it.

```
msum :: (Foldable t, MonadPlus m) => t (m a) -> m a
sum :: (Foldable t, Num a)         => t a -> a
```

*f.* `liftM*`

*a.* `liftM` Essentially a `fmap`.

```
liftM :: Monad m => (a -> b) -> m a -> m b
```

Lifts a `function` into `monadic equivalent`.

*b.* `liftM2` `Monadic liftA2`.

```
liftM2 :: Monad m => (a -> b -> c) -> m a -> m a -> m c
```

Lifts `binary function` into `monadic equivalent`.

*f.* Comonad `Category C comonad` is a `monad` of `opposite category Cop`.

*g.* Kleisli arrow `Morphism` that while doing computation also adds `monadic-able structure`.

`a -> m b`

- a. \*
- Kleisli arrows
  - Kleisli morphism
  - Kleisli morphisms

- h. Kleisli composition [Composition](#) of [Kleisli arrows](#).

```
(<=<) :: Monad m => (b -> m c) -> (a -> m b) -> a -> m c infixr 1
;; compare
(.)   ::          (b ->  c ) -> (a ->  b ) -> a ->  c
```

Often used left-to-right version:

```
(>=>) :: Monad m => (a -> m b) -> (b -> m c) -> a -> m c
;; compare
(>>=) :: Monad m =>          m a -> (a -> m b)          -> m b
```

Which allows to replace [monadic bind](#) chain with [Kleisli composition](#).

```
f1 arg >>= f2 >>= f3
==
f1 >=> f2 >=> f3 $ arg
==
f3 <=< f2 <=< f1 $ arg
```

- i. Kleisli category [Category](#)  $\mathcal{C}$ ,  $E$ ,  $\overrightarrow{\eta}$ ,  $\overrightarrow{\mu}$  [monad](#) over  $\mathcal{C}$ .

[Kleisli category](#)  $\mathcal{C}_T$  of  $\mathcal{C}$ :

$$\text{Obj}(\mathcal{C}_T) = \text{Obj}(\mathcal{C})$$

$$\text{Hom}_{\mathcal{C}_T}(x, y) = \text{Hom}_{\mathcal{C}}(x, E(y))$$

- j. Special monad

- a. Identity monad Wraps data in the [Identity constructor](#).

Useful: Creates [monads](#) from [monad](#) transformers.

[Bind](#): Applies internal value to the [bound function](#).

Code:

```
newtype Identity a = Identity { runIdentity :: a }

-- coerce is a function that directly moves data between type aliases
instance Functor Identity where
    fmap      = coerce

instance Applicative Identity where
```

```

pure      = Identity
(<*>)     = coerce

instance Monad Identity where
  m >=> k  = k (runIdentity m)

```

Example:

```

-- derive the State monad using the StateT monad transformer
type State s a = StateT s Identity a

```

- b. Maybe monad Something that may not be or not return a result. Any lookups into the real world, database queries.

**Bind:** **Nothing** input gives **Nothing** output, **Just x** input uses **x** as input to the **bound function**.

When some computation results in **Nothing** - drops the chain of computations and returns **Nothing**.

**Zero:** **Nothing**

Plus: result in first occurrence of **Just** else **Nothing**.

Code:

```

data Maybe a = Nothing | Just a

instance Monad Maybe where
  return      = Just
  fail        = Nothing
  Nothing >=> _ = Nothing
  (Just x) >=> f = f x

```

```

instance MonadPlus Maybe where
  mzero      = Nothing
  Nothing 'mplus' x = x
  x 'mplus' _    = x

```

Example:

Given 3 dictionaries:

- a. Full names to email addresses,
- b. Nicknames to email addresses,
- c. Email addresses to email preferences.

Create a **function** that finds a person's email preferences based on **either** a full name or a nickname.

```

data MailPref = HTML | Plain
data MailSystem = ...

getMailPrefs :: MailSystem -> String -> Maybe MailPref

```

```
getMailPrefs sys name =
  do let nameDB = fullNameDB sys
      nickDB = nickNameDB sys
      prefDB = prefsDB sys
      addr <- (lookup name nameDB) `mplus` (lookup name nickDB)
      lookup addr prefDB
```

- c. Either monad When computation results in **Left** - drops other computations & returns the received **Left**.
- d. Error monad Something that can fail, throw **exceptions**.

The failure process records the description of a failure. **Bind function** uses successful values as input to the **bound function**, and passes failure information on without executing the **bound function**.

Useful:

Composing **functions** that can fail. Handle **exceptions**, create **error** handling **structure**.

**Zero**: empty **error**.

Plus: if first **argument** failed then execute second **argument**.

- e. List monad Computations which may return 0 or more possible results.

**Bind**: The **bound function** is **applied** to all possible values in the input **list** and the resulting lists are concatenated into **list** of all possible results.

Useful: Building computations from **sequences** of non-deterministic operations.

**Zero**: []

Plus: (++)

a. \*

[] monad

- f. Reader monad Creates a read-only shared environment for computations.

The **pure function** ignores the environment, while **»=** passes the inherited environment to both subcomputations.

Today it is defined through **ReaderT** transformer:

```
type Reader r = ReaderT r Identity -- equivalent to ((->) e), (e ->)
```

Old definition was:

```
newtype Reader e a = Reader { runReader :: (e -> a) }
```

For (e ->):

- **Functor** is (.)

```
fmap :: (b -> c) -> (a -> b) -> a -> c
fmap = (.)
```

- [Applicative](#):

- `pure` is `const`

```
pure :: a -> b -> a
pure x _ = x
```

- `(<*>)` is:

```
(<*>) :: (a -> b -> c) -> (a -> b) -> a -> c
(<*>) f g = \a -> f a (g a)
```

- [Monad](#):

```
(>>=) :: (a -> b) -> (b -> a -> c) -> a -> c
(>>=) m k = Reader $ \r ->
  runReader (k (runReader m r)) r
```

```
join :: (e -> e -> a) -> e -> a
join f x = f x x
```

```
runReader
  :: Reader r a -- the Reader to run
  -> r -- an initial environment
  -> a -- extracted final value
```

Usage:

```
data Env = ...
```

```
createEnv :: IO Env
createEnv = ...
```

```
f :: Reader Env a
f = do
  a <- g
  pure a
```

```
g :: Reader Env a
g = do
  env <- ask -- "Open the environment namespace into env"
  a <- h env -- give env to h
  pure a
```

```
h :: Env -> a
... -- use env and produce the result
```

```
main :: IO ()
main = do
  env <- createEnv
  a = runReader g env
  ...
```

In Haskell under normal circumstances impure [functions](#) should not directly call impure [functions](#).

`h` is an impure [function](#), and `createEnv` is impure [function](#), so they should have intermediary.

- g.* Writer monad Computations which accumulate [monoid](#) data to a shared Haskell storage.

So `*` is parametrized by [monoidal type](#).

Accumulator is maintained separately from the returned values.

Shared value modified through [Writer monad](#) methods.

`*` frees creator and code from manually keeping the track of accumulation.

**Bind:** The [bound function](#) is [applied](#) to the input value, [bound function](#) allowed to `<>` to the accumulator.

```
type Writer r = WriterT r Identity
```

Example:

```
f :: Monoid b => a -> (a, b)
f a = if _condition_
      then runWriter $ g a
      else runWriter do
        a1 <- h a
        pure a1
```

```
g :: Monoid b => Writer b a
g a = do
  tell _value1_ -- accumulator <> _value1_
  pure a -- observe that accumulator stored inside monad and only a main value needs to
```

```
h :: Monoid b => Writer b a
h a = do
  tell _value2_ -- accumulator <> _value_
  pure a
```

```
runWriter :: Writer w a -> (a, w) -- Unwrap a writer computation as a (result, accumulator)
-- The inverse of writer.
```

`WriterT`, `Writer` unnecessarily keeps the entire logs in the memory. Use `fast-logger` for logging.

- h.* State monad Computations that pass-over a state.

The [bound function](#) is [applied](#) to the input value to produce a state transition [function](#) which is [applied](#) to the input state.

[Pure](#) functional language cannot update values in place because it violates [referential transparency](#).

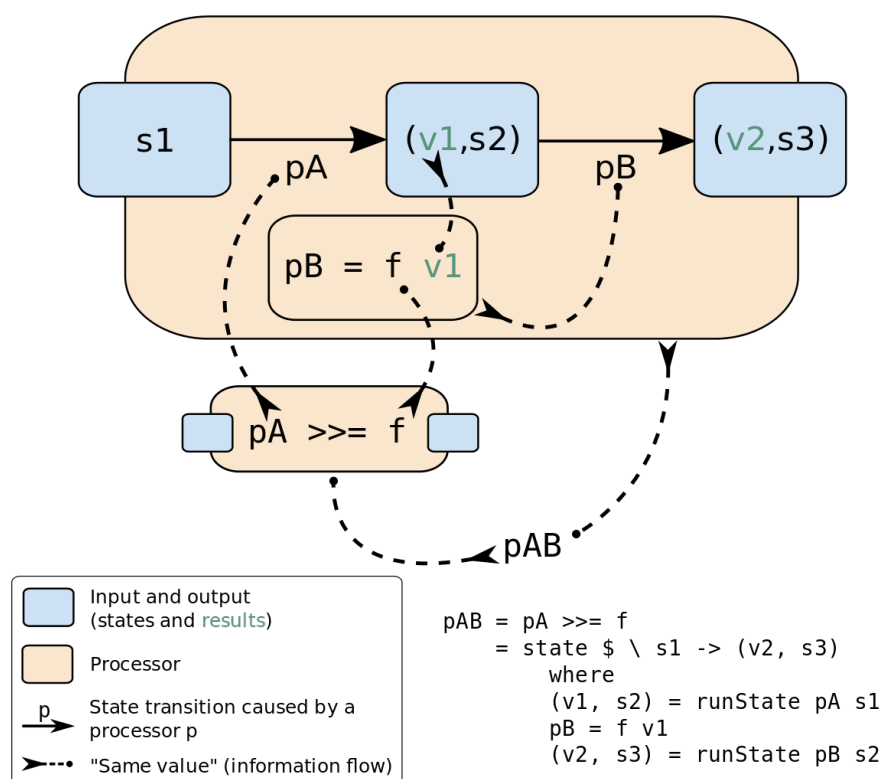
```
type State s = StateT s Identity
```

**Binding** copies and transforms the state **parameter** through the **sequence** of the **bound functions** so that the same state storage is never used twice. Overall this gives the illusion of in-place update to the programmer and in the code, while in fact the auto-generated transition **functions** handle the state changes.

Example **type**: `State st a`

**State** describes **functions** that consume a state and produce a **tuple** of result and an updated state.

**Monad** manages the state with the next process:



Where:

- $f$  - processor making **function**
- $pA, pAB, pB$  - state processors
- $sN$  - states
- $vN$  - values

**Bind** with a processor making **function** from state processor ( $pA$ ) creates a new state processor ( $pAB$ ).

The wrapping and unwrapping by `State/runState` is implicit.

- k. Monad transformer `*` is a practical solution to the current functional programming problem about [composition](#) of [monads](#).

[Monad](#) is not [closed](#) under composition.

[Composition](#) of [monadic types](#) does not always results in [monadic type](#).

Basic [case](#): during implementation of [monadic type composition](#), `type m T m a` arises, which does not allow to `unit`, `join` the `m` [monadic](#) layers.

`*` have desirable properties and can add them to [monads](#). `*` use their implementation to solve the composition [type](#) layering and allow to attach desirable [property](#) to result.

`*` solve [monad composition](#) and [type](#) layering by cheating, using own [structure](#) and information about itself. It is often that process involves a [catamorphism](#) of a `*` [type](#) layer.

In [type](#) signatures of transformers `*T m -> m` is already an extended [monad](#), so `*T` is just a wrapper to point that out.

Transformers have a light wrapper around the data that tags the modification with this transformer.

Main [monadic structure](#) `m` is wrapped around the internal data (core is `a`). The [structure](#) that corresponds to the transformer creation [properties](#) (if it emitted by  $\eta$  of a transformer), goes into `m`. Open [parameters](#) go external to the `m`.

```
newtype ExceptT e m a =
  ExceptT { runExceptT :: m (Either e a) }
```

```
newtype MaybeT m a =
  MaybeT { runMaybeT :: m (Maybe a) }
```

```
newtype ReaderT r m a =
  ReaderT { runReaderT :: r -> m a }
```

This has an [effect](#) that on stacking [monad](#) transformers, `m` becomes [monad stack](#), and every next transformer injects the transformer creation-specific properties  $\eta$  inside the [stack](#), so out-most transformer has inner-most [structure](#). Base [monad](#) is structurally the outermost.

- a. `MaybeT *` extends [monads](#) by injecting [Maybe](#) layer underneath [monad](#), and processing that [structure](#):

```
newtype MaybeT m a = MaybeT { runMaybeT :: m (Maybe a) }
```

- b. `EitherT *` extends [monads](#) by injecting [Either](#) layer underneath [monad](#), and processing that [structure](#):

```
newtype EitherT e m a = EitherT { runEitherT m (Either e a) }
```

`EitherT` of `either` package gets replaced by `ExceptT` of `transformers` or `mtl` packages.



a. \* ExceptT

c. ReaderT Definition:

```
newtype ReaderT r m a = ReaderT { runReaderT :: r -> m a }
```

\* **functions**: input **monad** `m a`, out: `m a` wrapped it in a free-variable `r` (**partially applied function**).  
That allows to use transformed `m a`, now it requires and can use the `r` passed environment.

To create a **Reader monad**:

```
type Reader r = ReaderT r Identity
```

d. MonadTrans **type class** Allows to **lift monadic** actions into a larger **context** in a neutral way.

**pure** takes a parametric **type** and embodies it into constructed **structure** (talking of **monad** transformers - **structure** of the stacked **monads**).

**lift** takes **monad** and extends it with a transformer.

In fact, for **monad** transformers - **lift** is a last stage of the **pure**, it follows from the **lift** law.

Method:

```
lift :: Monad m => m a -> t m a
```

**Lift** a computation from the **argument monad** to the constructed **monad**.

Neutral means:

```
lift . return = return
```

```
lift (m >>= f) = lift m >>= (lift . f)
```

The general pattern with **MonadTrans** instances is that it is usually lifts the **injection** of the known **structure** of transformer over some **Monad**.

**lift** embeds one **monadic** action into **monad transformer**.

The difference between **pure**, **lift** and **MaybeT** constructor becomes clearer if you look at the **types**:

Example, for **MaybeT IO a**:

```
pure      ::      a  -> MaybeT IO a
lift     ::  IO a  -> MaybeT IO a
MaybeT  ::  IO (Maybe a) -> MaybeT IO a
```

```
x = (undefined :: IO a)
```

```

:t (pure x)
(pure x) :: Applicative t => t (IO a) -- t recieves one argument of product type
:t (pure x :: MaybeT IO a)
-- Expected type: MaybeT IO a1
-- Actual type: MaybeT IO (IO a0)

-- While the real type would be
:t (pure x :: MaybeT IO (IO a))
(pure x :: MaybeT IO (IO a)) :: MaybeT IO (IO a) -- This goes into a conflict of what ty

:t (lift x)
(lift x) :: MonadTrans t => t IO a -- result is a proper expected product type

-- To belabour
:t (lift x :: MaybeT IO a)
(lift x) :: MonadTrans t => t IO a -- result is a proper expected product type

lift is a natural transformation  $\eta$  from an Identity monad (functor) with other monad
as content into transformer monad (functor), with the preservation of the contained
monad:

-- Abstract monads with content as parameters. Define '~>' as a family of morphisms that
type f ~> g = forall x. f x -> g x
-- follows
lift :: m ~> t m
a. MonadIO type class * - allows to lift IO action until reaching the IO monad layer
at the top of the Monad stack (which is allways in the Haskell code that does IO).

class (Monad m) => MonadIO m where
    liftIO :: IO a -> m a
liftIO laws:

liftIO . pure = pure

liftIO (m >>= f) = liftIO m >>= (liftIO . f)
Which is identical laws to MonadTrans lift.

Since lift is one step, and liftIO all steps - all steps defined in terms of one step
and all other steps, so the most frequent implementation is self-recursive lift .
liftIO:

liftIO ioa = lift $ liftIO ioa
a. *
    liftIO

```

### 3.5.7.7 Alternative type class

[Monoid](#) over [applicative](#). Has left catch [property](#).

Allows to run simoltaneously several instances of a computation (or computations) and from them yeld one result by law from  $\langle | \rangle :: \text{Type} \rightarrow \text{Type} \rightarrow \text{Type}$ .

Minimal complete definition:

```
empty :: f a    -- The identity element of <|>
(<|>) :: f a -> f a -> f a    -- Associative binary operation

a. *
    Alternative
```

### 3.5.7.8 «<=\*>=>»>

Do calculation, but ignore the value from the first [argument](#).

`* > ≡ >>`

## 3.5.8 Monoidal functor

[Functors](#) between [monoidal categories](#) that preserves [monoidal structure](#).

## 3.5.9 Fusion

```
fmap f . fmap g = fmap (f . g)
```

`*` - [functor](#) axiom that allows to greatly simplify computations.

### 3.5.10 «<=\$>=>»>

Get & [set](#) a value inside [Functor](#).

## 3.5.11 Multifunctor

Generalizes the concept of [functor](#) between [categories](#), canonical [morphisms](#) between multicategories.

Works over  $N$  [type](#) arguments instead of one.

To put simply - accepts multiple arguments, from that information constructs source [product category](#) ([Cartesian product](#)) of [categories](#), and is a [functor](#) from [product category](#) to target [category](#).

To put even simpler - [functor](#) that takes as an [argument](#) the [product](#) of [types](#).

In Haskell there is only one [category](#), [Hask](#), so in Haskell `*` is still  $(Hask \times Hask) \rightarrow Hask \Rightarrow |(Hask \times Hask) \equiv Hask| \Rightarrow Hask \rightarrow Hask$  [endofunctor](#).

Any [product](#) or sum in a Cartesian [category](#) is a `*`.

Code definition:

```

class Bifunctor f
  where
    bimap :: (a -> a') -> (b -> b') -> f a a' -> f a a'
    bimap f g = first f . second g
    first :: (a -> a') -> f a b -> f a' b
    first f = bimap f id
    second :: (b -> b') -> f a b -> f a b'
    second = bimap id

```

### 3.5.11.1 \*

Bifunctor

### 3.5.12 \*

«<=<\$=>>>

## 3.6 Hask category

Category of Haskell where objects are types and morphisms are functions.

It is a hypothetical category at the moment, since undefined and bottom values break the theory, is not Cartesian closed, it does not have sums, products, or initial object, () is not a terminal object, monad identities fail for almost all instances of the Monad class.

That is why Haskell developers think in subset of Haskell where types do not have bottom values. This only includes functions that terminate, and typically only finite values. The corresponding category has the expected initial and terminal objects, sums and products, and instances of Functor and Monad really are endofunctors and monads.

Hask contains subcategories, like Lst containing only list types.

Haskell and Category concepts:

- Things that take a type and return another type are type constructors.
- Things that take a function and return another function are higher-order functions.

### 3.6.1 \*

Hask

## 3.7 Magma

Set with a binary operation which form a closure.

### 3.7.1 Mag category

The [category of magmas](#), denoted *Mag*, has as [objects](#) - [sets](#) with a [binary operation](#), and [morphisms](#) given by homomorphisms of operations (in the [universal algebra](#) sense).

#### 3.7.1.1 \*

MAG  
Magma category  
Category of magmas

### 3.7.2 Semigroup

[Magma](#) with [associative property](#) of [operation](#).

Defined in Haskell as:

```
class Semigroup a where
  (<>) :: a -> a -> a
```

#### 3.7.2.1 \*

Semigroups

#### 3.7.2.2 Monoid

[Semigroup](#) with [identity](#) element. [Category](#) with a one [object](#).

Ideally fits as an accumulation class.

```
class Monoid m where
  mempty :: m
  mappend :: m -> m -> m
  mappend = (<>)
  mconcat :: [m] -> m
  mconcat = foldr mappend mempty
```

\* can be simplified to [category](#) with a single [object](#), remember that [monoid operation](#) is a [composition](#) of [morphisms operation](#) in [category](#).

For example to represent the whole non-negative integers with the one [object](#) and [morphism](#) "1" is absolutely enough, [composition operation](#) is "+".

```
import Data.Monoid
do
  show (mempty :: Num a => Sum a)
  -- "Sum {getSum = 0}"
  show $ Sum 1
  -- "Sum {getSum = 1}"
  show $ (Sum 1) <> (Sum 1) <> (Sum 1)
  -- "Sum {getSum = 3}"
  -- ...
```

Also backwards - any single-object category is a monoid. Category has an identity requirement and associativity of composition requirement, which makes it a free monoid.

a. \*

Monoidal  
Monoids

b. Monoid laws

a. Monoid left identity law

`mempty <> x = x`

b. Monoid right identity law

`x <> mempty = x`

c. Monoid associativity law

`x <> mempty = x (y <> z) = (x <> y) <> z`

`mconcat = foldr (mempty <>)`

Everything associative can be `mappend`.

c. Commutative monoid Commutativity property:

$x \circ y = y \circ x$

Opens a big abilities in concurrent and distributed processing.

a. \*

Abelian monoid

d. Group Monoid that has inverse for every element.

a. \*

Groups

b. Commutative group Group operation obeys the axiom of commutativity.

a. \*

Abelian group

b. Ring Commutative group under  $+$  & monoid under  $\times$ ,  $+$   $\times$  connected by distributive property.

- and  $\times$  are generalized binary operations of addition and multiplication.  $\times$  has no requirement for commutativity.

Example: set of same size square matrices of numbers with matrix operations form a ring.

a. \*

Rings

## 3.8 Morphism

*morphe* form

Arrow between two objects in a category.

General description: Arrow from source to target. Denotes something.

On a level of objects: is probably structure-preserving map from one mathematical structure to another of the same type.

Morphism is a generalization ( $f(x * y) \equiv f(x) \diamond f(y)$ ) of homomorphism ( $f(x * y) \equiv f(x) * f(y)$ ). Under morphism almost always is the meaning of homomorphism-like properties.

Morphism can be anything.

If morphism corresponds to function requirements - then it is a function.

### 3.8.1 \*

Morphisms

Arrow

Arrows

### 3.8.2 Homomorphism

*homos* same (was chosen because of initial English mistranslation to "similar")

*morphe* form

similar form

\* map between two algebraic structures of the same type, operation-preserving.

$$f_{x \rightarrow y} = f(a \star b) = f(a) \diamond f(b),$$

where  $x, y$  are sets with additional algebraic structure that includes  $\star, \diamond$  accordingly;  $a, b$  are elements of set  $x$ .

\* sends identity morphisms to identity morphisms and inverses to inverses.

The concept of \* has been generalized under the name of morphism to many structures that either do not have an underlying set, or are not algebraic.

#### 3.8.2.1 \*

Homomorphic

### 3.8.3 Identity morphism

Identity morphism - or simply identity:  $x \in C : id_x = 1_x : x \rightarrow x$

Composed with other morphism gives same morphism.

Corresponds to Reflexivity and Automorphism.

**3.8.3.1 Identity**

**Identity** only possible with **morphism**. See **Identity morphism**.

There is also distinct **Zero** value.

- a. Two-sided identity of a predicate  $P(e, a) = P(a, e) = a \mid \exists e \in S, \forall a \in S$   
 $P()$  is **commutative**.

**Predicate**

- b. Left identity of a predicate  $\exists e \in S, \forall a \in S : P(e, a) = a$

**Predicate**

- c. Right identity of a predicate  $P(a, e) = a \mid \exists e \in S, \forall a \in S$

**Predicate**

**3.8.3.2 Identity function**

Return itself.

$(\backslash x.x)$

$\text{id} :: a \rightarrow a$

**3.8.4 Monomorphism**

*mono* only

*morphe* form

Maps one to one (uniquely), so invertable (always has **inverse morphism**), so preserves the information/**structure**.

**Domain** can be equal or less to the **codomain**.

$f^{X \rightarrow Y}, \forall x \in X \exists! y = f(x) \models f(x) \equiv f_{mono}(x)$  - from **homomorphism context**

$f_{mono} \circ g1 = f_{mono} \circ g2 \models g1 \equiv g2$  - from general **morphism context**

Thus  $*$  is left cancelable.

If  $*$  is a **function** - it is **injective**. Initial **set** of  $f$  is fully uniquely mapped onto the **image** of  $f$ .

**3.8.4.1 \***

Monomorphic

Monomorphisms



### 3.8.5 Epimorphism

*epi* on, over  
*morphe* form

\* is right cancelable [morphism](#).

$f^{X \rightarrow Y}, \forall y \in Y \exists f(x) \models f(x) \equiv f_{epi}(x)$  - from [homomorphism context](#)

$g_1 \circ f_{epi} = g_2 \circ f_{epi} \Rightarrow g_1 \equiv g_2$  - from general [morphism context](#)

In [Set category](#) if \* is a [function](#) - it is [surjective](#) ([image](#) of it fully uses [codomain](#))  
[Codomain](#) is called a projection of the [domain](#).

\* fully maps into the target.

#### 3.8.5.1 \*

Epimorphic  
 Epimorphisms

### 3.8.6 Isomorphism

*isos* equal  
*morphe* form

Not equal, but equal for current intents and purposes.

[Morphism](#) that has [inverse](#).

Almost equal, but not quite: ([Integer](#), [Bool](#)) & ([Bool](#), [Integer](#)) - but can be transformed losslessly into one another.

[Bijective homomorphism](#) is also [isomorphism](#).

$$f^{-1, b \rightarrow a} \circ f^{a \rightarrow b} \equiv 1^a, f^{a \rightarrow b} \circ f^{-1, b \rightarrow a} \equiv 1^b$$

2 reasons for non-[isomorphism](#):

- [function](#) at least ones collapses a values of [domain](#) into one value in [codomain](#)
- [image](#) (of a [function](#) in [codomain](#)) does not fill-in [codomain](#). Then [isomorphism](#) can exists for [image](#) but not whole [codomain](#).

[Categories](#) are [isomorphic](#) if there  $R \circ L = ID$

#### 3.8.6.1 \*

Isomorphic  
 Isomorphisms

#### 3.8.6.2 Lax

Holds up to [isomorphism](#).

(upon the transformation can be used as the same)

### 3.8.7 Endomorphism

*endo* internal  
*morphe* form

Arrow from [object](#) to itself.

[Endomorphism](#) forms a [monoid](#) ([object](#) exists and [category](#) requirements already in place).

#### 3.8.7.1 Automorphism

*αυτο* *auto* self  
*form* form

[Isomorphic endomorphism](#).

Corresponds to [identity](#), [reflexivity](#), [permutation](#).

*a.* \*

Automorphic  
 Automorphisms

#### 3.8.7.2 \*

Endomorphic  
 Endomorphisms

### 3.8.8 Catamorphism

*kata* downward  
*morphe* form

Unique [arrow](#) from an initial [algebra structure](#) into different [algebra structure](#).

\* in FP is a generalization folding, deconstruction of a [data structure](#) into more primitive [data structure](#) using a [functor](#) F-[algebra structure](#).

\* reduces the [structure](#) to a lower level [structure](#).

\* creates a projection of a [structure](#) to a lower level [structure](#).

#### 3.8.8.1 \*

Catamorphic  
 Catamorphisms

#### 3.8.8.2 Catamorphism law

*a.* Hylomorphism [catamorphism](#)  $\circ$  [anamorphism](#)

Expanding and collapsing the [structure](#).

Table 3.2: [Catamorphism](#) laws in Haskell

Rule name	Haskell
cata-cancel	<code>cata phi . InF = phi . fmap (cata phi)</code>
cata-refl	<code>cata InF = id</code>
cata-fusion	<code>f . phi = phi . fmap f =&gt; f . cata phi = cata phi</code>
cata-compose	<code>eps :: f ~&gt; g =&gt; cata phi . cata (In . eps) = cata (phi . eps)</code>

*a.* \*

Hylomorphic  
Hylomorphisms

### 3.8.8.3 Anamorphism

Generalizes `unfold`.

Dual concept to [catamorphism](#).

Increases the [structure](#).

[Morphism](#) from a [coalgebra](#) to the final [coalgebra](#) for that [endofunctor](#).

Is a [function](#) that generates a [sequence](#) by repeated [application](#) of the [function](#) to its previous result.

*a.* \*

Anamorphic  
Anamorphisms

### 3.8.9 Kernel

[Kernel](#) of a [homomorphism](#) is a number that measures the [degree homomorphism](#) fails to meet [injectivity](#) (AKA be [monomorphic](#)).

It is a number of [domain](#) elements that fail [injectivity](#):

- elements not included into [morphism](#)
- elements that collapse into one element in [codomain](#)

thou [Kernel](#)  $[x|x \leftarrow 0||x \geq 2]$ .

Denotation:

$$\ker T = \{\mathbf{v} \in V : T(\mathbf{v}) = \mathbf{0}_W\}.$$

#### 3.8.9.1 Kernel homomorphism

[Morphism](#) of elements from the [kernel](#).

Complementary [morphism](#) of elements that make main [morphism](#) not [monomorphic](#).

### 3.9 Set category

Category in which **objects** are **sets**, **morphisms** are **functions**.

Denotation:

*Set*

### 3.10 Natural transformation

Roughly  $*$  is:

`trans :: F a -> G a`

, while `a` is **polymorphic variable**.

**Naturality** condition:  $\forall a \exists (F a \rightarrow G a)$ , or , analogous to **parametric polymorphism** in **functions**. Since  $*$  in a **category**, stating  $\forall (F a \rightarrow G a)$

**Naturality** condition means that all **morphisms** that take part in **homotopy** of source **functor** to target **functor** must exist, and that is the same, diagrams that take part in transformation, should commute, and different paths brings same result: if  $\alpha$  - **natural transformation**,  $\alpha_a$  **natural transformation component** -  $G f \circ \alpha_a = \alpha_b \circ F f$ .

Since  $*$  are just a **type** of parametric **polymorphic function** - they can **compose**.

$*$  ( $\vec{\eta}^{\mathcal{D}}$ ) is transforming :  $\vec{\eta}^{\mathcal{D}} \circ F^{C \rightarrow \mathcal{D}} = G^{C \rightarrow \mathcal{D}}$ .

$*$  **abstraction** creates higher-language of **Category** theory, allowing to talk about the **composition** and transformation of complex entities.

It is a process of transforming  $F^{C \rightarrow \mathcal{D}}$  into  $G^{C \rightarrow \mathcal{D}}$  using existing **morphisms** in target **category**  $\mathcal{D}$ .

Since it uses **morphisms** - it is **structure-preserving** transformation of one **functor** into another. It's mostly a lossy transformation. Only existing **morphisms** can make it exist.

Existence of  $*$  between two **functors** can be seen as some **relation**.

Can be observed to be a "**morphism of functors**", especially in **functor category**.

$*$  by  $\vec{\eta}_{yc}^{\mathcal{D}}((x, y)^{\mathcal{C}}) \circ F^{C \rightarrow \mathcal{D}}((x, y)^{\mathcal{C}}) = G^{C \rightarrow \mathcal{D}}((x, y)^{\mathcal{C}}) \circ \vec{\eta}_{xc}^{\mathcal{D}}((x, y)^{\mathcal{C}})$ , often written short  $\vec{\eta}_b \circ F(\vec{f}) = G(\vec{f}) \circ \vec{\eta}_a$ .

Notice that the  $\vec{\eta}_{xc}^{\mathcal{D}}((x, y)^{\mathcal{C}})$  depends on **objects&morphisms** of  $\mathcal{C}$ .

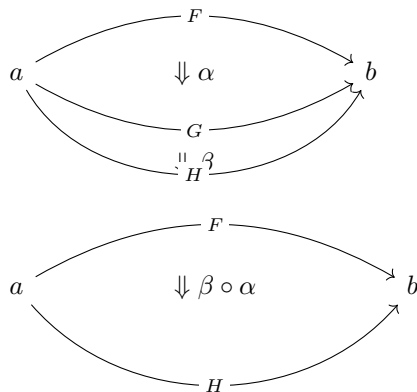
In words:  $*$  depends on  $F$  and  $G$  **functors**, ability of  $\mathcal{D}$  **morphisms** to do a **homotopy** of  $F$  to  $G$ , and  $*$ :

- for every **object** in  $\mathcal{C}$  picks **natural transformation component** in  $\mathcal{D}$ .
- for every **morphism** in  $\mathcal{C}$  picks the **commuting diagram** in  $\mathcal{D}$ , called naturality square.

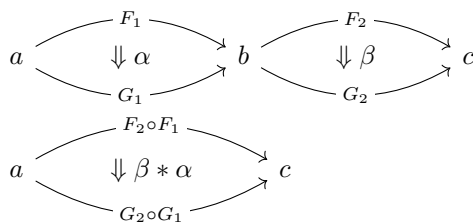
Also see: **Natural transformation in Haskell**

Knowledge of  $*$  forms a **2-category**.

Can be [composed](#) "vertically":



And horizontally, aka "Godement [product](#)":



[Compositions](#) can be done in any right [order](#), they abide the exchange law.

### 3.10.1 \*

Natural transformations  
 Naturality condition  
 Naturality

### 3.10.2 Natural transformation component

$$\vec{\eta}^{\mathcal{D}}(x) = F^{\mathcal{D}}(x) \rightarrow G^{\mathcal{D}}(x) \mid x \in \mathcal{C}$$

#### 3.10.2.1 \*

Component of natural transformation

### 3.10.3 Natural transformation in Haskell

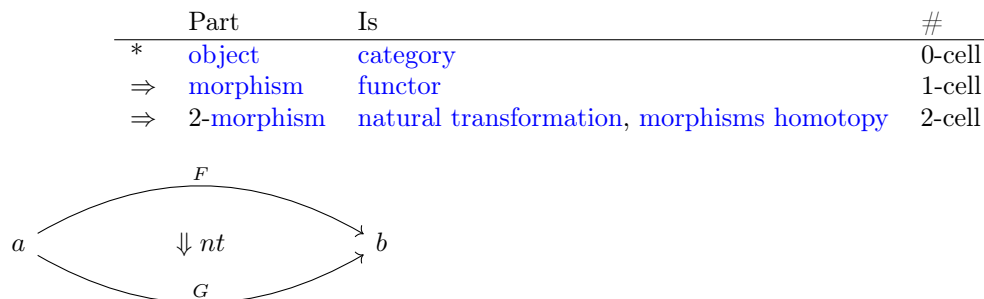
Family of [parametric polymorphism functions](#) between [endofunctors](#).

In [Hask](#) is  $F\ a \rightarrow G\ a$ . Can be analogued to repackaging data into another container, never modifies the [object](#) content, it only if - can delete it, because [operation](#) is lossy.

Can be sees as ortogonal to [functors](#).

### 3.10.4 Cat category

Category where:



Is Cartesian [closed category](#).

#### 3.10.4.1 \*

Cat

2-category

#### 3.10.4.2 Bicategory

2-category that is [enriched](#) and [lax](#).

For handling relaxed [associativity](#) - introduces associator, and for [identity](#) 1-left/right unitor.

Forming from bicategories higher [categories](#) by stacking levels of [abstraction](#) of such [categories](#) - leads to explosion of special cases, differences of every level, and so overall difficulties.

Stacking groupoids ([category](#) in which are [morphisms](#) are invertable) is much more homogenous up to infinity, and forms base of the [homotopy type](#) theory.

## 3.11 Category dual

[Category duality](#) behaves like a logical [inverse](#).

[Inverse](#)  $\mathcal{C} = \mathcal{C}^{op}$  - inverts the direction of [morphisms](#).

[Composition](#) accordingly changes to the [morphisms](#):  $(g \circ f)^{op} = f^{op} \circ g^{op}$

Any [statement](#) in the terms of  $\mathcal{C}$  in  $\mathcal{C}^{op}$  has the [dual](#) - the logical [inverse](#) that is true in  $\mathcal{C}^{op}$  terms.

Opposite preserves [properties](#):

- [products](#):  $(\mathcal{C} \times \mathcal{D})^{op} \cong \mathcal{C}^{op} \times \mathcal{D}^{op}$
- [functors](#):  $(F^{\mathcal{C} \rightarrow \mathcal{D}})^{op} \cong F^{\mathcal{C}^{op} \rightarrow \mathcal{D}^{op}}$
- [slices](#):  $(\mathcal{F} \downarrow \mathcal{G})^{op} \cong (\mathcal{G}^{op} \downarrow \mathcal{F}^{op})$

a. \*

Opposite category  
 Opposite categories  
 Category duality  
 Duality  
 Dual category  
 Dual

### 3.11.1 Coalgebra

Structures that are dual (in the category-theoretic sense of reversing arrows) to unital associative algebras.

Every coalgebra, by vector space duality, reversing arrows - gives rise to an algebra. In finite dimensions, this duality goes in both directions. In infinite - it should be determined.

## 3.12 Thin category

$\forall$  Hom sets contain zero or one morphism.

$$f \equiv g \mid \forall x, y \forall f, g : x \rightarrow y$$

A proset (preordered set).

### 3.12.1 \*

Proset category  
 Prosetal category  
 Poset category  
 Posetal category

## 3.13 Commuting diagram

Establishes equality in morphisms that have same source and target.

Draws the morphisms that are:

$$f = g \Rightarrow \{f, y\} : X \rightarrow Y$$

### 3.13.1 \*

Diagram commutes  
 Commutes

## 3.14 Universal construction

Algorithm of constructing definitions in Category theory.

Specially good to translate properties/definitions from other theories (Set theory) to Categories.

Method:

- a. Define a pattern that you defining.
- b. Establish ranking for pattern matches.
- c. The top of ranking, the best match or [set](#) of matches - is the thing you was looking for. Matches are [isomorphic](#) for defined rules.

\* uses Yoneda lemma, and as such constructions are defined until [isomorphism](#), and so [isomorphic](#) between each-other.

### 3.14.1 \*

Universal constructions

## 3.15 Product

[Universal construction](#):

$$\begin{array}{ccccc} & & c' & & \\ & p \swarrow & \downarrow ! & \searrow q & \\ a & \xleftarrow{\pi_a} & c & \xrightarrow{\pi_b} & b \end{array}$$

Pattern:  $p : c \rightarrow a, q : c \rightarrow b$

Ranking:  $\max \sum^{\vee} (! : c' \rightarrow c \mid p' = p \circ !, q' = q \circ !)$

$c'$  is another candidate.

For [sets](#) - [Cartesian product](#).

\* is a pair. Corresponds to [product data type](#) in [Hask](#) (inhabited with all elements of the [Cartesian product](#)).

[Dual](#) is [Coproduct](#).

### 3.15.1 \*

Products

## 3.16 Coproduct

Universal constructuon:

$$\begin{array}{ccccc} & & c' & & \\ & p \nearrow & \uparrow ! & \nwarrow q & \\ a & \xrightarrow{\iota_a} & c & \xleftarrow{\iota_b} & b \end{array}$$

Pattern:  $i : a \rightarrow c, j : b \rightarrow c$

Ranking:  $\max \sum^{\vee} (! : c \rightarrow c' \mid i' = ! \circ i, j' = ! \circ j)$

$c'$  is another candidate.



For **sets** - Disjoint union.

\* is a **set** assembled from other two **sets**, in Haskell it is a tagged **set** (analogous to disjoint union).

**Dual** is **Product**.

### 3.16.1 \*

Coproducts

## 3.17 Free object

General particular **structure**.

In which **structure**, **properties** autofollows from definition, axioms.

Also uses as a term when surcomstances of **structures**, rules, **properties**, axioms used coincide with the definition of a particular **object**  $\therefore$  form **object** of this **type** with the according **properties** and possibilities.

## 3.18 Internal category

**Category** which is included into a bigger **category**.

## 3.19 Hom set

All **morphisms** from source **object** to target **object**.

Denotation:

$$hom_C(X, Y) = (\forall f : X \rightarrow Y) = hom(X, Y) = C(X, Y)$$

Denotation was not standartized.

**Hom sets** belong to **Set category**.

In **Set category**:  $\exists!(a, b) \iff \exists! Hom, \forall Hom \in Set$ . **Set category** is special, **Hom sets** are also **objects** of it.

**Category** can include **Set**, and **hom sets**, or not.

### 3.19.1 \*

Hom-set

Hom sets

### 3.19.2 Hom-functor

$$hom : \mathcal{C}^{op} \times \mathcal{C} \rightarrow Set$$

**Functor** from the **product** of  $\mathcal{C}$  with its **opposite category** to the **category** of **sets**.

Denotation variants:

$$H_A = \text{Hom}(-, A)$$

$$h_A = \mathcal{C}(-, A)$$

$$\text{Hom}(A, -) : \mathcal{C} \rightarrow \text{Set}$$

Hom-bifunctor:

$$\text{Hom}(-, -) : \mathcal{C}^{op} \times \mathcal{C} \rightarrow \text{Set}$$

### 3.19.3 Exponential object

Generalises the notion of [function set](#) to internal [object](#).

As also [hom set](#) to [internal hom objects](#).

Cartesian [closed](#) ([monoidal](#)) [category](#) strictly required, as  $*$  multiplication holds [composition](#) requirement:

$$\circ : \text{hom}(y, z) \otimes \text{hom}(x, y) \rightarrow \text{hom}(x, z)$$

Denotation:

$$b^a$$

[Universal construction](#):

$$\begin{array}{ccc} c & c \times a & \\ \vdots & \vdots & \searrow \\ u & u \times 1^a & \\ \downarrow & \downarrow & \\ b^a & b^a \times a \cdot \text{eval} & b \end{array}, \text{ where in Category: } b^a - \text{exponential object, } \times - \text{product bifunctor,}$$

$a$  - [argument](#) of  $*$ ,  $b$  - result,  $c$  - candidate,  $b^a \equiv (a \Rightarrow b) - *$ .

$*$   $b^a$  (also as  $(a \Rightarrow b)$ ) represent exponentiation of [cardinality](#) of  $\forall b^a$  possibilities.

#### 3.19.3.1 $*$

Function object

Internal hom

Exponential objects

Hom object

Hom objects

#### 3.19.3.2 Enriched category

Uses [Hom objects](#) ([exponential objects](#)), which do not belong into [Set category](#).

[Category](#) is no longer small, now may be called large.

$$\text{hom}(x, y) \in K.$$

Called:  $*$  over  $K$  (which holds [hom objects](#)).

$a.$   $*$  Enriched

Large category

# Chapter 4

## Data type

Set of values.

For **type** to have sense the values must share some sense, **properties**.

### 4.1 \*

Type

Types

Data types

### 4.2 Actual type

**Data type** recieved by ->**inferring**->compiling->execution.

### 4.3 Algebraic data type

Composite **type** formed by combining other **types**.

#### 4.3.1 \*

AlgDT

### 4.4 Cardinality

Number of possible implementations for a given **type** signature.

**Disjunction**, sum - adds **cardinalities**.

**Conjunction**, **product** - multiplies **cardinalities**.

#### 4.4.1 \*

Cardinalities

## 4.5 Data constant

\* - [constant](#) value; [nullary data constructor](#).

## 4.6 Data constructor

One instance that [inhabit data type](#).

## 4.7 data declaration

[Data type declaration](#) is the most general and versatile form to create a new [data type](#).  
Form:

```
data [context =>] type typeVars1..n
  = con1  c1t1..i
  | ...
  | conm  cmt1..q
  [deriving]
```

## 4.8 Dependent type

When [type](#) and values have [relation](#) between them. [Type](#) has restrictions for values, value of a [type variable](#) has a result on the [type](#).

### 4.8.1 \*

Dependent types

## 4.9 Gen type

[Generator](#). [Gen type](#) is to generate pseudo-random values for parent [type](#). Produces a [list](#) of values that gets infinitely cycled.

## 4.10 Higher-kinded data type

Any combination of \* and ->

[Type](#) that take more [types](#) as arguments.

*Humblly really a [function](#)*

### 4.10.1 \*

Higher-kinded data types

## 4.11 newtype declaration

Create a new [type](#) from old [type](#) by attaching a new [constructor](#), allowing [type class instance declaration](#).

```
newtype FirstName = FirstName String
```

Data will have exactly the same representation at runtime, as the [type](#) that is wrapped.

```
newtype Book = Book (Int, Int)
```

```
    (,)
    / \
Integer Integer
```

## 4.12 Principal type

The most generic [data type](#) that still [typechecks](#).

## 4.13 Product data type

Is an [algebraic data type](#) representation of a [product](#) construction.  
Formed by logical [conjunction](#) (AND, '[\\*](#)').

Haskell forms:

```
-- 1. As a tuple (the uncurried & most true-form)
(T1, T2)

-- 2. Curried form, data constructor that takes two types
C T1 T2

-- 3. Using record syntax. =r# <inhabitant>= would return the respective =T#=
C { r1 :: T1
    , r2 :: T2
    }
```

### 4.13.1 \*

Product type

### 4.13.2 Sequence

Enumerated (ordered) [set](#).

Denotation:

```
()
( , )
( , , )
( , , ... )
```

More general mathematical denotation was not established, variants:

$$(n)_{n \in \mathbb{N}}$$

$$\omega \rightarrow X$$

$$\{i : Ord \mid i < \alpha\}$$

In Haskell: [Data type](#) that stores multiple ordered values withing a single value.

Table 4.1: [Sequence constructor](#) naming by [arity](#)

Name	Arity	Denotation
<a href="#">Unit</a> , empty	0	()
<a href="#">Singleton</a>	1	( <u>_</u> )
Tuple, pair, two-tuple	2	( , )
Triple, three- <a href="#">tuple</a>	3	( , , )
<a href="#">Sequence</a>	N	( , , ... )

#### 4.13.2.1 \*

Sequences  
Tuples  
Ordered pair  
Ordered triple

#### 4.13.2.2 List

The same [type objects sequence](#).

Denotation:

[]  
[ , ]  
[ , , ]  
[ , , ... ]

Haskell definition:

```
data [] a = [] | a : [a]
```

Definition is self-referential (self-[recursive](#)), can be seen as [anamorphism](#) (unfold) of the [] (empty [list](#), memory cell which is container of particular [type](#)) and : ([cons operation](#), pointer). As such - can create non-terminating [data type](#) (and computation), in other words - infinite.

## 4.14 Proxy type

[Proxy type](#) holds no data, but has a phantom [parameter](#) of [arbitrary type](#) (or even [kind](#)). Able to provide [type](#) information, even though has no value of that [type](#) (or it can be may too costly to create one).

```
data Proxy a = ProxyValue
```

```
let proxy1 = (ProxyValue :: Proxy Int) -- a has kind 'Type'
let proxy2 = (ProxyValue :: Proxy List) -- a has kind 'Type -> Type'
```

## 4.15 Static typing

Typechecking takes place at compile level.

## 4.16 Structural type

Mathematical type. They form into structural type system.

### 4.16.1 \*

Structural

## 4.17 Structural type system

Strict global hierarchy and relationships of types and their properties.

Haskell type system is \*.

In most languages typing is name-based, not structural.

### 4.17.1 \*

Structural typing

## 4.18 Sum data type

Algebraic data type formed by logical disjunction (OR '|').

## 4.19 Type alias

Create new type constructor, and use all data structure of the base type.

## 4.20 Type class

Type system construct that adds a support of ad hoc polymorphism.

Type class makes a nice way for defining behaviour, properties over many types/objects at once.

### 4.20.1 \*

Type classes

Typeclass

Typeclasses

### 4.20.2 Arbitrary type class

Type class of `QuickCheck.Arbitrary` (that is reexported by `QuickCheck`) for creating a generator/distribution of values.

Useful function is `arbitrary` - that autogenerates values.

#### 4.20.2.1 Arbitrary function

Depends on `type` and generates values of that `type`.

### 4.20.3 CoArbitrary type class

Pseudogenerates a function basing on resulting `type`.

```
coarbitrary :: CoArbitrary a => a -> Gen b -> Gen b
```

#### 4.20.3.1 \*

`CoArbitrary`

### 4.20.4 Typeable type class

Allows dynamic `type` checking in Haskell for a `type`.

Shift a `typechecking` of `type` from compile time to runtime.

\* `type` gets wrapped in the universal `type`, that shifts the `type` checks to runtime.

Also allows:

- Get the `type` of something at runtime (ex. print the `type` of something `typeOf`).
- Compare the `types`.
- Reifying functions from `polymorphic type` to concrete (for functions like `:: Typeable a => a -> String`).

#### 4.20.4.1 \*

`Typeable`

### 4.20.5 Type class inheritance

Type class has a `superclass`.



### 4.20.6 Derived instance

Type class instances sometimes can be automatically [derived](#) from the parent [types](#).

Type classes such as Eq, Enum, Ord, Show can have instances generated based on definition of data type.

P.S.

Language options:

- DeriveAnyClass
- DeriveDataTypeable
- DeriveFoldable
- [DeriveFunctor](#)
- DeriveGeneric
- DeriveLift
- DeriveTraversable
- DerivingStrategies
- DerivingVia
- GeneralisedNewtypeDeriving
- StandaloneDeriving

#### 4.20.6.1 \*

Derived  
Deriving

## 4.21 Type constant

[Nullary type constructor](#).

## 4.22 Type constructor

Name of the [data type](#).

[Constructor](#) that takes [type](#) as an [argument](#) and produces new [type](#).

## 4.23 type declaration

Synonym for existing [type](#). Uses the same [data constructor](#).

```
type FirstName = String
```

Used to distinct one entities from other entities, while they have the same [type](#).  
Also main [type functions](#) can operate on a new [type](#).

## 4.24 Typed hole

`*` - is a `_` or `_name` in the [expression](#). On [evaluation](#) GHC would show the [derived type](#) information which should be in place of the `*`. That information helps to fill in the gap.

### 4.24.1 \*

Typed holes

## 4.25 Type inference

Automatic [data type](#) detection for [expression](#).

### 4.25.1 \*

Inferring  
Infer  
Infers  
Inferred

## 4.26 Type class instance

Block of implementations of [functions](#), based on unique [type class](#)->[type](#) pairing.

## 4.27 Type rank

Weak ordering of [types](#).

The rank of [polymorphic type](#) shows at what level of nesting `forall` [quantifier](#) appears.  
Count-in only [quantifiers](#) that appear to the left of [arrows](#).

```
f1 :: forall a b. a -> b -> a    ==    fi :: a -> b -> c
g1 :: forall a b. (Ord a, Eq b) => a -> b -> a    ==    g1 :: (Ord a, Eq b) => a -> b -> a
```

f1, g1 - [rank-1 types](#). Haskell itself implicitly adds universal [quantification](#).

```
f2 :: (forall a. a->a) -> Int -> Int
g2 :: (forall a. Eq a => [a] -> a -> Bool) -> Int -> Int
```

f2, g2 - [rank-2 types](#). Quantifier is on the left side of a  $\rightarrow$ . Quantifier shows that [type](#) on the left can be overloaded.

[Type inference](#) in Rank-2 is possible, but not higher.

```
f3 :: ((forall a. a->a) -> Int) -> Bool -> Bool
```

f3 - [rank3-type](#). Has [rank-2 types](#) on the left of a  $\rightarrow$ .

```
f :: Int -> (forall a. a -> a)
g :: Int -> Ord a => a -> a
```

f, g are rank 1. [Quantifier](#) appears to the right of an [arrow](#), not to the left. These [types](#) are not Haskell-98. They are supported in [RankNTypes](#).

#### 4.27.1 \*

Type ranks  
Rank type  
Rank types  
Rank-1 type  
Rank-1 types  
Rank-2 type  
Rank-2 types  
Rank-3 type  
Rank-3 types

## 4.28 Type variable

Refer to an unspecified [type](#) in Haskell [type](#) signature.

## 4.29 Unlifted type

[Type](#) that directly exist on the hardware. The [type abstraction](#) can be completely removed. With [unlifted types](#) Haskell [type](#) system directly manages data in the hardware.

#### 4.29.1 \*

Unlifted types

## 4.30 Data structure

### 4.30.1 Cons cell

Cell that values may [inhabit](#).

### 4.30.2 Construct

`(:) :: a -> [a] -> [a]`

#### 4.30.2.1 \*

Cons

### 4.30.3 Leaf

-

### 4.30.4 Node

\*  
/ \

## 4.31 Linear type

[Type](#) system and [algebra](#) that also track the multiplicity of data.  
There are 3 general [linear type](#) groups:

- 0 - exists only at [type level](#) and is not allowed to be used at value level. Aka `s` in [ST-Trick](#).
- 1 - data that is not duplicated
- 1< - all other data, that can be duplicated multiple times.

### 4.31.1 \*

Linear types

## 4.32 NonEmpty list data type

Data.[List](#).NonEmpty

Has a [Semigroup](#) instance but can't have a [Monoid](#) instance. It never can be an empty [list](#).

```
data NonEmpty a = a :| [a]
  deriving (Eq, Ord, Show)
```

`:|` - an [infix](#) data constructor that takes two ([type](#)) arguments. In other words `:|` returns a [product type](#) of left and right

### 4.33 Session type

\* - allows to check that behaviour conforms to the protocol.

So far very complex, not very productive (or well-established) topic.

### 4.34 Binary tree

```
data BinaryTree a
  = [[Leaf]]
  | [[Node]] (BinaryTree a) a (BinaryTree a)
```

### 4.35 Bottom value

A `_` non-value in the [type](#) or [pattern match expression](#). Placeholder for anything.

```
-- _ fits *.
```

#### 4.35.1 \*

Bottom  
Bottom values

### 4.36 Bound

Haskell `*` [type class](#) means to have lowest value & highest value, so a [bounded](#) range of values.

#### 4.36.1 \*

Bounded

### 4.37 Constructor

*a.* [Type constructor](#)

*b.* [Data constructor](#)

Also see: [Constant](#)

#### 4.37.1 \*

Constructors

## 4.38 Context

Type constraints for polymorphic variables.

Written before the main `type` signature, denoted:

```
TypeClass a => ...
```

### 4.38.1 \*

Contexts

## 4.39 Inhabit

Values that is a component of `data type set`.

## 4.40 Maybe

```
data Maybe
  = Nothing
  | Just a
```

Does not represent the information why `Nothing` happened.

For `error` - use `Either`.

Do not propagate `*`.

Handle `*` locally to `where` it is produced. `Nothing` does not hold useful info for debugging & short-circuits the processes. Do not expect code `type` being bug-free, do not return `Maybe` to end user since it would be impossible to debug, return something that preserves `error` information.

### 4.40.0.1 \*

Nodes

## 4.41 Expected type

`Data type inferred` from the text of the code.

## 4.42 ADT

a. Abstract data type

b. Algebraic data type

## 4.43 Concrete type

Fully defined `type`. Non-polymorphic type.

## 4.44 Type punning

When [type constructor](#) and [data constructor](#) have the same name.

Theoretically if person knows the rules - \* can be solved, because in Haskell [type](#) and [data declaration](#) have different places of use.

## 4.45 Kind

[Kind](#) -> [Type](#) -> [Data](#)

### 4.45.1 \*

Kinds

## 4.46 IO

[Type](#) for values whose evaluations has a possibility to cause side effects or return unpredictable result.

Haskell standard uses [monad](#) for constructing and transforming [IO](#) actions.

[IO](#) action can be evaluated multiple times.

[IO data type](#) has unpure imperative actions inside. Haskell is [pure Lambda calculus](#), and un-pure [IO](#) integrates in the Haskell purely ([type](#) system abstracts [IO](#) unpurity inside [IO data type](#)).

[IO sequences effect](#) computation one after another in [order](#) of needed computation, or occurrence:

```
twoBinds :: IO ()
twoBinds =
  putStrLn "First:" >>
  getLine >>=
  \a ->
  putStrLn "Second:" >>
  getLine >>=
  \b ->
  putStrLn ("\nFirst: "
    ++ a ++ ".\nSecond "
    ++ b ++ ".")
main = twoBinds
```

Sequencing is achieved by compilation of effects performing only while they receive the sugared-in & passed around the `RealWorld` fake [type](#) value, that value in the every computation gets the new "value" and then passed to the next requested computation. But special thing is about this [parameter](#), this `RealWorld` [type](#) value passed, but never looked at. GHC realizes, since value is never used, - it means value and [type](#) can be equated to `()` and moreover reduced from the code, and sequencing stays.

## Chapter 5

# Declaration

[Binding](#) name to [expression](#).



## Chapter 6

# Differential operator

Denotation.

$\frac{d}{dx}$ ,  $D$ ,  $D_x$ ,  $\partial_x$ .

Last one is partial.

$e^{t\frac{d}{dx}}$  - Shift operator.

### 6.1 \*

Differential

## Chapter 7

# Dispatch

Send, transmission, reference.

## Chapter 8

# Effect

Observable action.

## Chapter 9

# Evaluation

For FP see [Bind](#).

## Chapter 10

# Expression

Finite combination of symbols that is well-formed according to rules that depend on the [context](#).

### 10.1 \*

Expressions

### 10.2 Closed-form expression

\* - mathematical [expression](#) that can be evaluated in a finite number of operations.

May contain:

- constants
- [variables](#)
- operations (e.g.,  $+$   $\times$   $\div$ )
- [functions](#) (e.g., nth root, exponent, logarithm, trigonometric [functions](#), and [inverse](#) hyperbolic [functions](#)), but usually no limit.

### 10.3 RHS

Right-hand side of the [expression](#).

### 10.4 LHS

Left-hand side of the [expression](#).

## 10.5 Redex

Reducible expression.

## 10.6 Concatenate

Link together sequences, expressions.

## 10.7 Alpha equivalence

Equivalence of a processes in expressions. If expressions have according parameters different, but the internal processes are literally the same process.

## 10.8 Ground expression

Expression that does not contain any free variables.

### 10.8.1 \*

Ground formula

## 10.9 Variable

A name for expression.

Haskell has immutable variables.

Except when you hack it with explicit funtions.

### 10.9.1 \*

Variables

## Chapter 11

# First-class

Means *it*:

- Can be used as value.
- Passed as an [argument](#).

From 1&2 -> *it* can include itself.

# Chapter 12

## Function

Full dependency of one quantity from another quantity.

Denotation:

$$y = f(x)$$

$$f : X \rightarrow Y,$$

where  $X$  is domain,  $Y$  is codomain.

Directionality and property of invariability emerge from one another.

```
-- domain func codomain
*          ->   *
```

$$y(x) = (zx^2 + bx + 3 \mid b = 5)$$

$\backslash$ Parameter  $\backslash$ Free variable  $\backslash$ Bound variable  $\backslash$ Var  $\backslash$ Constant

$\backslash$ Name of the function

Lambda abstraction is a function.

Function is a mathematical operation.

Function = Total function = Pure function. Function theoretically can be to memoized.

Also see:

Partial function

Inverse function - often partially exists (partial function).

### 12.1 \*

Functions

Bound variable



## 12.2 Arity

Number of [parameters](#) of the [function](#).

- [nullary](#) -  $f()$
- [unary](#) -  $f(x)$
- [binary](#) -  $f(x,y)$
- [ternary](#) -  $f(x,y,z)$
- [n-ary](#) -  $f(x,y,z..)$

## 12.3 Bijection

[Function](#) is a complete one-to-one pairing of elements of [domain](#) and [codomain](#) ([image](#)).  
It means [function](#) both [surjective](#) (so [image](#) == [codomain](#)) and [injective](#) (every [domain](#) element has unique correspondence to the [image](#) element).

For [bijection inverse](#) always exists.

[Bijection operation](#) holds the [equivalence](#) of [domain](#) and [codomain](#).

Denotation:

>->>  
f : X Y

L<sup>A</sup>T<sub>E</sub>X needed to combine symbols:  
 $f : X \mapsto Y$

Corersponds to [isomorphism](#).

### 12.3.1 \*

Bijjective  
Bijjective function

## 12.4 Combinator

[Function](#) without free [variables](#).  
[Higher-order function](#) that uses only [function application](#) and other combinators.

```
\a -> a
\ a b -> a b
\f g x -> f (g x)
\f g x y -> f (g x y)
```

Not combinators:

```
\ xs -> sum xs
```

Informal broad meaning: referring to the style of organizing libraries centered around the idea of combining things.

## 12.5 Function application

\* - [bind](#) the [argument](#) to the [parameter](#) of a [function](#), and do a [beta-reduction](#).

### 12.5.1 \*

Apply  
Applied  
Applying  
Application

## 12.6 Function body

[Expression](#) that haracterizes the process.

## 12.7 Function composition

```
(.) :: (b -> c) -> (a -> b) -> a -> c
```

```
a -> (a -> b) -> (b -> c) -> c
```

In Haskell inline [composition](#) requires:

```
h.g.f $ i
```

### 12.7.1 \*

Composition  
Compose  
Composed

## 12.8 Function head

Is a part with name of the [function](#) and it's paramenterers.  
AKA:  $f(x)$

## 12.9 Function range

The range of a [function](#) refers to [either](#) the [codomain](#) or the [image](#) of the [function](#), depending upon usage. Modern usage almost always uses range to mean [image](#).  
So, see [Function image](#).

## 12.10 Higher-order function

[Function arity](#) > 1.

—

Has [function](#) as a [parameter](#).  
Evaluates to [function](#).

### 12.10.1 \*

HOF

### 12.10.2 Fold

[Catamorphism](#) of a [structure](#) to a lower [type](#) of [structure](#). Often to a single value.

[\\*](#) is a [higher-order function](#) that takes a [function](#) which operates with both main [structure](#) and accumulator [structure](#), [\\*](#) applies units of [data structure](#) to a [function](#) which works with accumulator. Upon traversing the whole [structure](#) - the accumulator is returned.

## 12.11 Injection

[Function](#) one-to-one injects from [domain](#) into [codomain](#).

Keeps distinct pairing of elements of [domain](#) and [image](#).  
Every element in [image](#) corresponds to one element in [domain](#).

$$\forall a, b \in X, f(a) = f(b) \Rightarrow a = b$$

$$\exists(\text{inverse function}) \mid \forall(\text{injective function})$$

Denotion:

$\rightarrow$

$f : X \rightarrow Y$

$f : X \rightarrow Y$

Corresponds to [Monomorphism](#).

**12.11.1   \***

Injective  
Injective function  
Injectivity

**12.12   Partial function**

One that does not cover all [domain](#).  
[Unsafe](#) and causes trouble.

**12.13   Purity**

\* means properly abstracted.

If the contrary - [abstraction](#) is unpure.

Also see: [pure function](#).

**12.13.1   \***

Pure

**12.14   Pure function**

[Function](#) that is [pure](#)  $\equiv$  [referentially transparent function](#).

**12.15   Sectioning**

Writing [function](#) in a parentheses. Allows to pass around [partially applied functions](#).

**12.16   Surjection**

[Function](#) uses [codomain](#) fully.

$$\forall y \in Y, \exists x \in X$$

Denotation:  
 $f : X \twoheadrightarrow Y$

Corresponds to [Epimorphism](#).

**12.16.1   \***

Surjective  
Surjective function

**12.17   Unsafe function**

**Function** that does not cover at least one edge **case**.

**12.17.1   \***

Unsafe

**12.18   Variadic**

\* - having **variable arity** (often up to indefinite).

**12.19   Domain**

Source **set** of a **function**.  
 $X$  in  $X \rightarrow Y$ .

**12.20   Codomain**

$Y$  in  $X \rightarrow Y$ .  
**Codomain** - target **set** of a **function**.

**12.21   Open formula**

Logical **function** that has **arity** and produces **proposition**.

**12.22   Recursion**

Repeated **function application** when sometimes same **function** gets called.

Allows computation that may require indefinite amount of work.

**12.22.1   \***

Recursive

**12.22.2 Base case**

A part of a [recursive function](#) that trivially produces result.

**12.22.3 Tail recursion**

Tail calls are [recursive](#) invocantions of itself.

**12.22.4 Polymorphic recursion**

[Type](#) of the [parameter](#) changes in [recursive](#) invocations of [function](#).

Is always a higher-ranked [type](#).

**12.22.4.1 \***

Milner–Mycroft typability

Milner–Mycroft calculus

**12.23 Free variable**

[Variable](#) in the fuction that is not [bound](#) by the head.

Until there are \* - [function](#) stays [partially applied](#).

**12.24 Closure**

$f(x) = f^{\mathcal{X} \rightarrow \mathcal{X}} \mid \forall x \in \mathcal{X}, \mathcal{X}$  is [closed](#) under  $f$ , it is a trivial [case](#) when [operation](#) is legitimate for all values of the [domain](#).

[Operation](#) on members of the [domain](#) always produces a members of the [domain](#). The [domain](#) is [closed](#) under the [operation](#).

In the [case](#) when there is a [domain](#) values for which [operation](#) is not legitimate/not exists:

$f(x) = f^{\mathcal{V} \rightarrow \mathcal{X}} \mid \mathcal{V} \in \mathcal{X}, \forall x \in \mathcal{V}, \mathcal{X}$  is [closed](#) under  $f$ .

**12.24.1 \***

Closed

**12.25 Parameter**

*para* subsidiary

*metron* measure

Named variable of a [function](#).

[Argument](#) is a supplied value to a [function parameter](#).

[Parameter](#) ([formal parameter](#)) is an [irrefutable](#) pattern, and implemented that way in Haskell.

### 12.25.1 \*

Parameters

Formal parameter

Formal parameters

## 12.26 Partial application

Part of [function parameters](#) applied.

### 12.26.1 \*

Partially applied

## 12.27 Well-formed formula

[Expression](#), logical [function](#) that is/can produce a [proposition](#).

### 12.27.1 \*

Well formed formula

WFF

wff

WFFS

wffs

# Chapter 13

## Homotopy

homós same

One can be "continuously deformed" into the other.

For example - [functions](#), [functors](#).

[Natural transformation](#) is a [homotopy](#) of [functors](#).

### 13.1 \*

Homotopies

Homotopic



## Chapter 14

# Lambda calculus

Universal model of computation. Which means  $\lambda$  can implement any [Turing machine](#).  
Based on [function abstraction](#) and [application](#) by substituting [variables](#) and [binding](#) values.

$\lambda$  has [lambda terms](#):

- [variable](#) ( $x$ )
- [application](#) ( $(ts)$ )
- [abstraction](#) ([lambda function](#)) ( $(\lambda x.t)$ )

### 14.1 $\lambda$

Lambda term  
Lambda terms  
Lambda variable  
Lambda variables

### 14.2 Lambda cube

[-cube](#) shows the 3 dimensions of generalizations from simply typed [Lambda calculus](#) to [Calculus of constructions](#).

Each dimension of the cube corresponds to a new way of making [objects](#) depend on other [objects](#):

- ([First-class polymorphism](#)) - terms allowed to depend on [types](#), corresponding to [polymorphism](#).
- ([Higher-rank polymorphism](#)) - [types](#) depending on terms, corresponding to [dependent types](#).
- ([Type class](#)) - [types](#) depending on [types](#), corresponding to [type](#) operators.

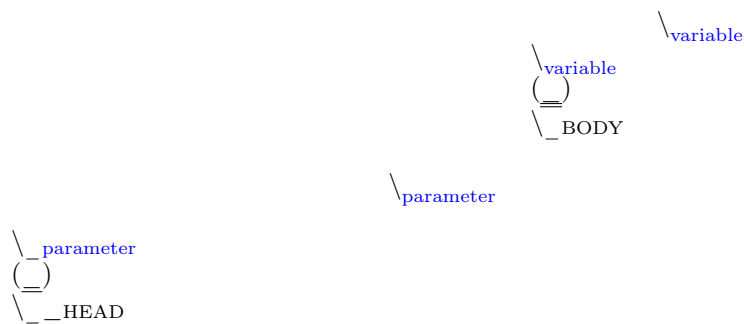
**14.2.1 \***

-cube  
 $\lambda$ -cube

**14.3 Lambda function**

Function of Lambda calculus.

$\lambda xy.x^2 + y^3$   
 $\wedge \wedge \wedge \wedge$

**14.3.1 \***

Lambda abstraction

**14.3.2 Anonymous lambda function**

Lambda function that is not binded to any name.

**14.3.2.1 \***

Anonymous lambda function

**14.4  $\beta$ -reduction**

Equation of a parameter to a bound variable, then reducing parameter from the head.

**14.4.1 \***

$\beta$  reduction  
 Beta-reduction  
 Beta reduction

**14.4.2  $\beta$ -normal form**

No beta reduction is possible.

**14.4.2.1 \***

$\beta$  normal form  
 Beta normal form  
 Beta-normal form

**14.5 Calculus of constructions**

Extends the Curry–Howard correspondence to the proofs in the full intuitionistic [predicate](#) calculus (includes proofs of [quantified statements](#)).

[Type](#) theory, typed programming language, and constructivism (philosophy) foundation for mathematics.

Directly relates to Coq programming language.

**14.5.1 \***

«<CoC>»

**14.6 Curry–Howard correspondence**

[Equivalence](#) of {[First-order logic](#), computer programming, [Category](#) theory}. They represent each-other, possible in one - possible in the other, so all the definitions and theorems have analogues in other two.

Gives a ground to the [equivalence](#) of computer programs and mathematical proofs.

Lambek added analogue to Cartesian [closed category](#), which can be used to model logic and [type](#) theory.

Table 14.1: Table of basic correspondence

Logic	Type	Category
True	() (any inhabited <a href="#">type</a> )	Terminal
False	<a href="#">Void</a>	Initial
$a \wedge b$	(a, b)	$a \times b$
$a \vee b$	<a href="#">Either</a> a b	$a / b$
$a \Rightarrow b$	$a \rightarrow b$	$b^a$

**14.6.1 \***

Curry–Howard isomorphism  
 Curry–Howard–Lambek

**14.7 Currying**

Translating the [evaluation](#) of a multiple [argument function](#) (or a [tuple](#) of arguments) into evaluating a [sequence](#) of [functions](#), each with a single [argument](#).

**14.7.1 \***

Curry

**14.8 Hindley–Milner type system**

Classical [type](#) system for the [Lambda calculus](#) with [Parametric polymorphism](#) and [Type inference](#). [Types](#) marked as [polymorphic variables](#), which enables [type inference](#) over the code.

**14.8.1 \***

Hindley–Milner

Damas–Milner

Damas–Hindley–Milner

**14.9 Reduction**

Take out something from a [structure](#), make simpler.

See [Beta reduction](#)

**14.9.1 \***

Reducible

**14.10  $\beta$ - $\eta$  normal form**

All  [\$\beta\$ -reduction](#) and  [\$\eta\$ -abstraction](#) are done in the [expression](#).

**14.10.1 \***

beta-eta normal form

beta eta normal form

**14.11  $\eta$ -abstraction**

$$(\lambda x. Mx) \xleftarrow{\eta} M$$

```
\ x -> g . f $ x
```

```
\ x -> g . f      --eta-abstraction
```

**14.11.1    \***

$\eta$ -reduction  
 $\eta$ -conversion  
 $\eta$  abstraction  
 $\eta$  reduction  
 $\eta$  conversion  
eta-abstraction  
eta-reduction  
eta-conversion  
eta abstraction  
eta reduction  
eta conversion

**14.12    Lambda expression**

See [Lambda calculus](#) ([Lambda terms](#)) and [Expression](#). In majority cases meaning some [Lambda function](#).

## Chapter 15

# Lense

Library of combinators to provide Haskell (functional language without mutation) with the emulation of **get**-ters and **set**-ters of imperative language.

# Chapter 16

## Operation

Calculation into output value. Can have [zero](#) & more inputs.

### 16.1 Constant

[Nullary operation](#).

Also see: [Type constant](#).

### 16.2 Binary operation

$\forall(a, b) \in S, \exists P(a, b) = f(a, b) : S \times S \rightarrow S$

#### 16.2.1 \*

Binary operations

### 16.3 Operator

Denotation symbol/name for the [operation](#).

#### 16.3.1 Shift operator

[Shift operator](#) defined by Lagrange through [Differential operator](#).

$$T^t = e^{t \frac{d}{dx}}$$

##### 16.3.1.1 \*

Shift

## 16.4 Infix

Form of writing of **operator** or **function** in-between **variables** for **application**.

For priorities see **Fixity**.

## 16.5 Fixity

Declares the presedence of action of a **function/operator**.

Funciton **application** has presedence higher then all **infix** operators/**functions** (virtually giving it a **priority** 10).

Table 16.1: Haskell operators **priority** and **fixity** association

P	L	Non	R
10			F.A.
9	!!		.
8			^ ^ ^ **
7	*/ div		
6	+-		
5			: , ++
4		<comparison> elem	
3			&&
2			OR
1			
0			\$ \$! seq

### 16.5.1 \*

Infixl

Infixr

Priority

Precedence

## 16.6 Zero

\* is the value with which **operation** always yelds **Zero** value.

$zero, n \in C : \forall n, zero * n = zero$

\* is distinct from **Identity** value.



## Chapter 17

# Permutation

Bijjective function from domain to itself.

Domain & permutation functions & function composition form a group.

## Chapter 18

# Phrase

Composable expression.

## Chapter 19

# Point-free

Paradigm [where function](#) only describes the [morphism](#) itself.

Process of converting [function](#) to [point-free](#).

If brackets `()` can be changed to `$` then `$` equal to [composition](#):

```
\ x -> g (f x)
\ x -> g $ f x
\ x -> g . f $ x
\ x -> g . f      --eta-abstraction
```

```
\ x1 x2 -> g (f x1 x2)
\ x1 x2 -> g $ f x1 x2
\ x1 x2 -> g . f x1 $ x2
\ x1      -> g . f x1
```

### 19.1 \*

Pointfree  
Tacit  
Tacit programming

### 19.2 Blackbird

`(.) . (.) :: (b -> c) -> (a1 -> a2 -> b) -> a1 -> a2 -> c`

[Composition of compositions](#) `(.) . (.)`. Allows to [compose](#)-in a [binary function](#) `f1(c) (.) . (.) f2(a,b)`.

```
\ f g x y -> f (g x y)
```

#### 19.2.1 \*

```
.) .
(.) . (.)
Composition of compositions
```

### 19.3 Swing

```
swing :: (((a -> b) -> b) -> c -> d) -> c -> a -> d
swing = flip . (. flip id)
swing f = flip (f . runCont . return)
swing f c a = f ($ a) c
```

### 19.4 Squish

```
f >>= a . b . c =<< g
```

## Chapter 20

# Polymorphism

*polis* many

At once several forms.

In Haskell - [abstract](#) over [data types](#).

\* [types](#):

### 20.1 \*

Polymorphic

### 20.2 Levity polymorphism

Extending [polymorphism](#) to work with unlifted and lifted [types](#).

### 20.3 Parametric polymorphism

[Abstracting](#) over [data types](#) by [parameter](#).

*In most languages named as 'Generics' (generic programming).*

[Types](#):

#### 20.3.1 Rank-1 polymorphism

[Parametric polymorphism](#) in [rank-1 types](#) by [type variables](#).

##### 20.3.1.1 \*

Prenex

Prenex polymorphism

### 20.3.2 Let-bound polymorphism

It is [property](#) chosen for Haskell [type](#) system.

Haskell is based on [Hindley-Milner type](#) system, it is [let-bound](#).

To have strict [type inference](#) with `*` - if `let` and `where` declarations are [polymorphic](#) -  $\lambda$  declarations - should be not.

See: [Good: In Haskell parameters bound by lambda declaration instantiate to only one concrete type](#).

### 20.3.3 Constrained polymorphism

Constrained [Parametric polymorphism](#).

#### 20.3.3.1 Ad hoc polymorphism

Artificial [constrained polymorphism](#) dependent on incoming [data type](#).

It is [interface dispatch](#) mechanism of [data types](#).

Achieved by creating a [type class instance functions](#).

*Commonly known as overloading.*

a. `*`

Ad-hoc polymorphism

Ad hoc polymorphic

Ad-hoc polymorphic

Constraint

Constraints

### 20.3.4 Impredicative polymorphism

`*` allows [type](#) entities with [polymorphic types](#) that can contain [type](#) itself.

$T = \forall X. X \rightarrow X : T \in X \models T \in T$

The most powerful form of [parametric polymorphism](#).

See: [Impredicative](#).

This approach has Girard's paradox ([type systems](#) [Russell's paradox](#)).

#### 20.3.4.1 `*`

First-class polymorphism

### 20.3.5 Higher-rank polymorphism

Means that [polymorphic types](#) can appear within other [types](#) ([types of function](#)).

There is a cases [where higher-rank polymorphism](#) than the a Ad hoc - is needed. For example [where ad hoc polymorphism](#) is used in [constraints](#) of several different implementations of [functions](#), and you want to build a [function](#) on top - and use the [abstract interface](#) over these [functions](#).

```
-- ad-hoc polymorphism
f1 :: forall a. MyType Class a => a -> String    ==    f1 :: MyType Class a => a -> String
f1 = -- ...

-- higher-rank polymorphism
f2 :: Int -> (forall a. MyType Class a => a -> String) -> Int
f2 = -- ...
```

By moving forall inside the [function](#) - we can achieve [higher-rank polymorphism](#).

From: <https://news.ycombinator.com/item?id=8130861>

Higher-rank polymorphism is formalized using System F, and there are a few implementations of (in  
Useful example aslo a [ST-Trick monad](#).

### 20.3.5.1 \*

Rank-n polymorphism

## 20.4 Subtype polymorphism

Allows to declare usage of a [Type](#) and all of its Subtypes.

T - [Type](#)

S - Subtype of [Type](#)

<: - subtype of

$S <: T = S \leq T$

Subtyping is:

If it can be done to T, and there is subtype S - then it also can be done to S.

$S <: T : f^{T \rightarrow X} \Rightarrow f^{S \rightarrow X}$

## 20.5 Row polymorphism

Is a lot like [Subtype polymorphism](#), but aligns itself on allowance (with | r) of subtypes and [types](#) with requested [properties](#).

```
printX :: { x :: Int | r } -> String
printX rec = show rec.x

printY :: { y :: Int | r } -> String
printY rec = show rec.y

-- type is inferred as '{x :: Int, y :: Int | r } -> String'
printBoth rec = printX rec ++ printY rec
```

## 20.6 Kind polymorphism

Achieved using a phantom [type argument](#) in the [data type declaration](#).

```
;;          * -> *
data Proxy a = ProxyValue
```

Then, by default the [data type](#) can be inhabited and fully work being partially defined.  
But multiple instances of [kind polymorphic type](#) can be distinguished by their particular [type](#).

Example is the [Proxy type](#):

```
data Proxy a = ProxyValue

let proxy1 = (ProxyValue :: Proxy Int) -- * :: Proxy Int
let proxy2 = (ProxyValue :: Proxy a)   -- * -> * :: Proxy a
```

## 20.7 Linearity polymorphism

Leverages [linear types](#).

For example - if [fold](#) over a dynamic array:

- a. In basic Haskell - array would be copied at every step.
- b. Use low-level [unsafe functions](#).
- c. With [Linear type function](#) we guarantee that the array would be used only at one place at a time.

So, if we use a [function](#) `(* -o * -o -o *)` in `foldr` - the [fold](#) will use the initial value only once.



# Chapter 21

## Pragma

**Pragma** - instruction to the compiler that specifies how a compiler should process the code. **Pragma** in Haskell have form:

```
{-# PRAGMA options #-}
```

### 21.1 LANGUAGE pragma

Controls what variations of the language are permitted.

It has a **set** of allowed options: [https://downloads.haskell.org/~ghc/latest/docs/html/users\\_guide/glasgow\\_exts.html](https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/glasgow_exts.html), which can be supplied.

#### 21.1.1 LANGUAGE option

##### 21.1.1.1 \*

Language options

##### 21.1.1.2 Useful by default

```
import EmptyCase
import FlexibleContexts
import FlexibleInstances
import InstanceSigs
import MultiParamTypeClasses
```

##### 21.1.1.3 AllowAmbiguousTypes

Allow **type** signatures which appear that they would result in an unusable **binding**. However GHC will still check and complain about a **functions** that can never be called.

##### 21.1.1.4 ApplicativeDo

Enables an **alternative** in-depth **reduction** that translates the do-notation to the operators `<$>`, `<*>`, `join` as far as possible.

For GHC to pickup the patterns, the final **statement** must match one of these patterns exactly:

```
pure E
pure $ E
return E
return $ E
```

When the [statements](#) of [do expression](#) have dependencies between them, and [ApplicativeDo](#) cannot [infer](#) an [Applicative type](#) - GHC uses a heuristic  $O(n^2)$  algorithm to try to use `<*>` as much as possible. This algorithm usually finds the best solution, but in rare complex cases it might miss an opportunity. There is also  $O(n^3)$  algorithm that finds the optimal solution: `-foptimal-applicative-do`.

Requires `ap = <*>`, `return = pure`, which is true for the most [monadic types](#).

- Allows use of `do`-notation with [types](#) that are an instance of [Applicative](#) and [Functor](#)
- In some [monads](#), using the [applicative](#) operators is more efficient than [monadic bind](#). For example, it may enable more parallelism.

The only way it shows up at the source level is that you can have a `do expression` with only [Applicative](#) or [Functor](#) constraint.

It is possible to see the actual translation by using `-ddump-ds`.

#### 21.1.1.5 ConstrainedClassMethods

Enable the definition of further [constraints](#) on individual class methods.

#### 21.1.1.6 CPP

Enable [C preprocessor](#).

#### 21.1.1.7 DeriveFunctor

Automatic [deriving](#) of instances for the [Functor type class](#).

For [type power set functor](#) is unique, its derivation implementation can be autochecked.

#### 21.1.1.8 ExplicitForAll

Allow explicit [forall](#) quantifier in places [where](#) it is implicit by Haskell.

#### 21.1.1.9 FlexibleContexts

Ability to use complex [constraints](#) in class [declaration contexts](#).

The only restriction on the [context](#) in a class [declaration](#) is that the class hierarchy must be acyclic.

```
class C a where
  op :: D b => a -> b -> b

class C a => D a where ...
```

$C \Rightarrow D$ , so in `C` we can talk about `D`.

Synergizes with [ConstraintKinds](#).

#### 21.1.1.10 FlexibleInstances

Allow [type class](#) instances [types](#) contain nested [types](#).

```
instance C (Maybe Int) where ...
```

Implies [TypeSynonymInstances](#).

#### 21.1.1.11 GeneralizedNewtypeDeriving

Enable GHC's `newtype` cunning generalised [deriving](#) mechanism.

```
newtype Dollars = Dollars Int
  deriving (Eq, Ord, Show, Read, Enum, Num, Real, Bounded, Integral)
```

(In old Haskell-98 only `Eq`, `Ord`, `Enum` could be inherited.)

#### 21.1.1.12 ImplicitParams

Allow definition of [functions](#) expecting implicit [parameters](#). In the Haskell that has static scoping of [variables](#) allows the dynamic scoping, such as in classic Lisp or ELisp. Sure thing this one can be puzzling as hell inside Haskell.

#### 21.1.1.13 LambdaCase

Enables [expressions](#) of the form:

```
\case { p1 -> e1; ...; pN -> eN }
```

-- OR

```
\case
  p1 -> e1
  ...
  pN -> eN
```

#### 21.1.1.14 MultiParamTypeClasses

Implies: [ConstrainedClassMethods](#)

Enable the definitions of [typeclasses](#) with more than one [parameter](#).

```
class Collection c a where
```

**21.1.1.15 MultiWayIf**

Enable multi-way-if syntax.

```
if | guard1 -> code1
   | ...
   | guardN -> codeN
```

**21.1.1.16 OverloadedStrings**

Enable overloaded string literals (string literals become desugared via the `IsString` class).

With overload, string literals has [type](#):

```
(IsString a) => a
```

The usual string syntax can be used, e.g. `ByteString`, `Text`, and other variations of string-like [types](#).

Now they can be used in pattern matches as `char->integer` translations. To [pattern match](#) `Eq` must be [derived](#).

To use class `IsString` - [import](#) it from `GHC.Ext`.

**21.1.1.17 PartialTypeSignatures**

Partial [type](#) signature contains [wildcards](#), placeholders (`_`, `_name`).

Allows programmer to which parts of a [type](#) to annotate and which to [infer](#). Also applies to [constraint](#) part.

As untuped [expression](#), partly typed can not polymorphically recurse.

[-Wno-partial-type-signatures](#) supresses [infer](#) warnings.

**21.1.1.18 RankNTypes**

Enable [types](#) of [arbitrary](#) rank.

See [Type rank](#).

Implies [ExplicitForAll](#).

Allows `forall` [quantifier](#):

- Left side of  $\rightarrow$
- Right side of  $\rightarrow$
- as [argument](#) of a [constructor](#)
- as [type](#) of a field

- as `type` of an implicit `parameter`
- used in pattern `type` signature of `lexically scoped type variables`

#### 21.1.1.19 ScopedTypeVariables

By default `type variables` do not have a `scope` except inside `type` signatures `where` they are used.

When there are internal `type` signatures provided in the code block (`where`, `let`, etc.) they (main `type` description of a `function` and internal `type` descriptions) restrain one-another and become not truly `polymorphic`, which creates a bounding interdependency of `types` that GHC would complain about.

\* option provides the `lexical scope` inside the code block for `type variables` that have `forall quantifier`. Because they are now lexically scoped - those `type variables` are used across internal `type` signatures.

For details see: <https://ocharles.org.uk/guest-posts/2014-12-20-scoped-type-variables.html>

Implies `ExplicitForAll`.

#### 21.1.1.20 TupleSections

Allow `tuple` section syntax:

```
(, True)
(, "I", , , "Love", , 1337)
```

#### 21.1.1.21 TypeApplications

Allow `type application` syntax:

```
read @Int 5

:type pure @[]
pure @[] :: a -> [a]

:type (<*>) @[]
(<*>) @[] :: [a -> b] -> [a] -> [b]

--

instance (CoArbitrary a, Arbitrary b) => Arbitrary (a -> b)

> ($ 0) <$> generate (arbitrary @(Int -> Int))
```

#### 21.1.1.22 TypeSynonymInstances

Now `type` synonym can have its own `type class` instances.

**21.1.1.23 UndecidableInstances**

Permit instances which may lead to [type-checker](#) non-termination.

GHC has Instance termination rules regardless of [FlexibleInstances](#) [FlexibleContexts](#).

**21.1.1.24 ViewPatterns**

```
foo (f1 -> Pattern1) = c1
foo (fn -> Pattern2 a b) = g1 a b
```

(*expression* → *pattern*): take what is came to match - [apply](#) the *expression*, then do *pattern*-match, and return what originally came to match.

[Semantics](#):

- *expression* & *pattern* share the [scope](#), so also [variables](#).

n (*expression* → *pattern*) t1.

- \* are like [pattern guards](#) that can be nested inside of other patterns.
- \* are a convenient way to pattern-match [algebraic data type](#).

Additional possible usage:

```
foo a (f2 a -> Pattern3 b c) = g2 b c -- only for function definitions
foo ((f,_), f -> Pattern4) = c2 -- variables can be bount to the left in data constructors and t
```

**21.1.1.25 DatatypeContexts**

Allow [contexts](#) in [data types](#).

```
data Eq a => Set a = NilSet | ConsSet a (Set a)
```

```
-- NilSet :: Set a
-- ConsSet :: Eq a => a -> Set a -> Set a
```

Considered misfeature. Deprecated. Going to be removed.

**21.1.1.26 StandaloneKindSignatures**

[Type](#) signatures for [type-level declarations](#).

```
type <name_1> , ... , <name_n> :: <kind>
```

```
type MonoTagged :: Type -> Type -> Type
data MonoTagged t x = MonoTagged x
```

```
type Id :: forall k. k -> k
type family Id x where
  Id x = x
```

```

type C :: (k -> Type) -> k -> Constraint
class C a b where
  f :: a b

type TypeRep :: forall k. k -> Type
data TypeRep a where
  TyInt    :: TypeRep Int
  TyMaybe :: TypeRep Maybe
  TyApp    :: TypeRep a -> TypeRep b -> TypeRep (a b)

```

< GHC 8.10.1 - [type](#) signatures were only for [term level](#) declarations.

Extension makes signatures feature more uniformal.

Allows to [set](#) the [order](#) of [quantification](#), [order](#) of [variables](#) in a [kind](#). For example when using [TypeApplications](#).

Allows to [set](#) full [kind](#) of derivable class, solving situations with [GADT](#) return [kind](#).

```

a. *
   SAKS
   Standalone kind signatures

```

#### 21.1.1.27 PartialTypeSignatures

Very helpful. Helps to solve [type level](#), helps to establish [type](#) signatures and [constraints](#).

Allow to provide `_` in the [type](#) signatures to automatically infer in the [type](#) information there.

Wild cards:

- [Type](#)

```
f :: _ -> _ -> a
```

- [Constraint](#)

```
f :: _ => a -> b -> c
```

- [Named](#)

```
f :: _x -> _x -> a
```

allows to identify the same [wildcard](#).

### 21.1.2 How to make a GHC LANGUAGE extension

In `libraries/ghc-boot-th/GHC/LanguageExtensions/Type.hs` add new [constructor](#) to the Extension [type](#)

```
data Extension
  = Cpp
  | OverlappingInstances
  ...
  | Foo
```

```
/main/DynFlags.hs extend xFlagsDeps:
```

```
xFlagsDeps = [
  flagSpec "AllowAmbiguousTypes" LangExt.AllowAmbiguousTypes,
  ...
  flagSpec "Foo"                  LangExt.Foo
]
```

It is for basic [case](#). For [testing](#), parser see further: <https://blog.shaynefletcher.org/2019/02/adding-ghc-language-extension.html>



## Chapter 22

# Predicative

Non-self-referencing definition.

—

*Antonym - [Impredicative](#).*

## Chapter 23

# Compositionality

Complex [expression](#) is determined by the constituent [expressions](#) and the rules used to combine them.

If the meaning fully obtainable from the parts and [composition](#) - it is full, [pure compositionality](#).

If there exists [composed idiomatic expression](#) - it is unfull, unpure [compositionality](#), because meaning leaks-in from the sources that are not in the [composition](#).

### 23.1 \*

Principle of compositionality

Composition

Compositional

## Chapter 24

### $\Psi$ -combinator

Transforms two of the same [type](#), [applying](#) same mediate transformation, and then transforming those into the result.

```
import Data.Function (on)
on :: (b -> b -> c) -> (a -> b) -> a -> a -> c

--\
  * ---
--/
```

#### 24.1 \*

Psi-combinator  
On-combinator

# Chapter 25

## Quantifier

Specifies the quantity of specimens.

Two most common [quantifiers](#)  $\forall$  ([Forall](#)) and  $\exists$  (Exists).  
 $\exists!$  - one and only one (exists only unique).

### 25.1 \*

Quantification  
Quantifiers  
Quantified

### 25.2 Forall quantifier

Permits to not [infer](#) the [type](#), but to use any that fits. The variant depends on the [LANGUAGE option](#) used:  
[ScopedTypeVariables](#)  
[RankNTypes](#)  
[ExistentialQuantification](#)

#### 25.2.1 \*

Forall

## Chapter 26

# Referential transparency

Given the same input return the same output.

So:

- \* **expression** can be replaced with its corresponding resulting value without change for program's behavior.

- \* **functions** are **pure**.

### 26.1 \*

Referentially transparent

## Chapter 27

# Relation

[Relationship](#) between two [objects](#).  
Subset of a [Cartesian product](#) between [sets](#) of [objects](#).  
Is not directed and not limited.

### 27.1 \*

Relations  
Relationship

## Chapter 28

# REPL

Read-eval-print loop, aka interactive shell.

# Chapter 29

## Semantics

Philosophical study of meaning.  
Meaning of symbols, words.

### 29.1 Operational semantics

Constructing proofs from logical [assertions](#) and verifying/checking/asserting things about execution and procedures their [properties](#), such as correctness, safety or security.

Good to solve in-point localized tasks.

Process of [abstraction](#).

### 29.2 Denotational semantics

Construction of [objects](#), that describe/tag the meanings. In Haskell often [abstractions](#) that are ment (denotations), implemented directly in the code, sometimes exist over the code - allowing to reason and implement.

\* are [composable](#).

Good to achieve more broad approach/meaning.

Also see [Abstraction](#).

#### 29.2.1 Abstraction

abs away from, off (in absentia)  
tractus draw, haul, drag

Purified generalization of process.

Forgetting the details ([axiomatic semantics](#)). Simplified approach. Out of sight - out of mind.

\* creates a new semantic level in which one can be absolutely precise ([operational semantics](#)).



It is a great did to name an [abstraction](#) ([denotational semantics](#)).

### 29.2.1.1 \*

Abstractions

Abstracting

Abstract

### 29.2.1.2 Leaky abstraction

[Abstraction](#) that leaks details that it is supposed to [abstract](#) away.

a. \*

Leaky abstractions

### 29.2.1.3 Object

Absolute [abstraction](#).

Point.

Can have [properties](#).

Often abstracts sometging, for example some [structure](#), [maybe](#) mathematical.

a. \*

Structure

Structures

Objects

b. Terminal object One that recieves unique [arrow](#) from every [object](#).

$$\exists ! : x \rightarrow 1 \mid \exists 1 \in \mathcal{C}, \forall x \in \mathcal{C}$$

\* is an empty [sequence](#) () in Haskell.

Called a [unit](#), so recieves *terminal* or [unit](#) arrow.

[Dual](#) of [initial object](#).

Denotation:

[Category](#) theory

1

Haskell

()

- c. Initial object One that emits unique [arrow](#) into every [object](#).

$$\exists! : \emptyset \rightarrow x \mid \exists \emptyset \in \mathcal{C}, \forall x \mathcal{C}$$

If [initial object](#) is `Void` (most frequently) - emitted [arrows](#) called absurd, because they can not be called.

[Dual](#) of [terminal object](#).

Denotation:

[Category](#) theory:

$\emptyset$

Haskell:

`Void`

### 29.2.2 Ambigram

ambi both

*grámma* written character

[Object](#) that from different points of view has the same meaning.

While this word has two contradictory diametrically opposite usages, one was chosen (more frequent).

But it has... Both.

*TODO: For merit of differentiating the meaning about different meaning referring to [Tensor](#) as [object](#) with many meanings.*

### 29.2.3 Binary

Two of something.

### 29.2.4 Arbitrary

*arbitrarius* uncertain

Random, any one of.

Used as: Any one with *this* [set](#) of [properties](#). ([constraints](#), [type](#), etc.).

When there is a talk about any [arbitrary](#) value - in fact it is a talk about the generalization of computations over the [set](#) of [properties](#).

### 29.2.5 Refutable

One that has an option to fail.

### 29.2.6 Irrefutable

One that can not fail.

## 29.3 Axiomatic semantics

Empirical process of studying something complex by finding and analyzing true [statements](#) about it.

Good for examining interconnections.

### 29.3.1 Property

Something has a [property](#) in the real world, and in theory its [property](#) corresponds to the law/laws, axioms.

In Haskell under [property](#)/law most often [properties](#) of [algebraic structures](#).

There [property testing](#) wich does what it says.

#### 29.3.1.1 \*

Properties

#### 29.3.1.2 Associativity

Joined with common purpose.

$$P(a, P(b, c)) \equiv P(P(a, b), c) \mid \forall (a, b, c) \in S,$$

\* - the operations can be grouped arbitrarily.

[Property](#) that determines how operators of the same [precedence](#) are grouped, (in computer science also in the absence of parentheses).

Etymology:

Latin *associatus* past participle of *associare* "[join with](#)", from assimilated form of *ad* "to" + *sociare* "unite with", from *socius* "companion, ally" from PIE *\*sokw-yo-*, suffixed form of root *\*sekw-* "to follow".

In Haskell \* has influence on parsing when compounds have same [fixity](#).

a. \*

Associative

Associative law

Associativity law

### 29.3.1.3 Left associative

\* - the operations are grouped from the left.

Example:

In lambda [expressions](#) same level parts follow grouping from left to right.

$(\lambda x.x)(\lambda y.y)z \equiv ((\lambda x.x)(\lambda y.y))z$

a. \*

Left associativity

Left-associative

### 29.3.1.4 Right associative

\* - the operations are grouped from the right.

### 29.3.1.5 Non-associative

Operations can't be chained.

Often is the [case](#) when the output [type](#) is incompatible with the input [type](#).

### 29.3.1.6 Basis

$\beta\alpha\sigma\iota\varsigma$  - stepping

The initial point, unreducible axioms and terms that spawn a theory.

AKA see [Category](#) theory, or Euclidian geometry [basis](#).

- a. Contravariant The [property](#) of [basis](#), in which if new [basis](#) is a [linear](#) combination of the prior [basis](#), and the change of [basis](#) inverse-proportional for the description of a [Tensors](#) in this basis.

Denotation:

Components for [contravariant basis](#) denoted in the upper indices:

$V^i = x$

The [inverse](#) of a [covariant](#) transformation is a [contravariant](#) transformation. Whenever a vector should be invariant under a change of [basis](#), that is to say it should represent the same geometrical or physical [object](#) having the same magnitude and direction as before, its components must transform according to the [contravariant](#) rule.

a. \*

Contravariant cofunctor

Contravariant functor - More inline term is [Contravariant cofunctor](#)

- b. Covariant The [property](#) of [basis](#), in which if new [basis](#) is a [linear](#) combination of the prior [basis](#), and the change of [basis](#) proportional for a descriptions of [tensors](#) in basis.

Denotation:

Components for [covariant basis](#) denoted in the upper indices:

$$V_i = x$$

a. \*

Covariant functor

Covariant cofunctor

### 29.3.1.7 Commutativity

$$\forall(a, b) \in S : P(a, b) \equiv P(b, a)$$

a. \*

Commutative

Commutative law

### 29.3.1.8 Idempotence

First [application](#) gives a result. Then same [operation](#) can be [applied](#) multiple times without changing the result.

Example: Start and Stop buttons on machines.

a. \*

Idempotent

Idempotency

### 29.3.1.9 Distributive property

Set S and two [binary](#) operators  $+$   $\times$ :

- $x \times (y + z) = (x \times y) + (x \times z)$  -  $\times$  is left-[distributive](#) over  $+$
- $(y + z) \times x = (y \times x) + (z \times x)$  -  $\times$  is right-[distributive](#) over  $+$
- left-&right-[distributive](#) -  $\times$  is [distributive](#) over  $+$

a. \*

Distributive rule

Distributive axiom

Distributive law

Distributive

## 29.4 Argument

*arguere* to make clear, to shine

\* - evidence, proof, [statement](#) that results in system consequences.

### 29.4.1 Argument of a function

A value binded to the [function parameter](#). Value/topic that the fuction would process/deal with.

Also see Argument.

#### 29.4.1.1 \*

Function argument

## 29.5 Content word

Words that name [objects](#) of reality and their qualities.

## 29.6 Ancient Greek and Latin prefixes

### 29.6.1 \*

Greek prefix

Latin prefix

## 29.7 Idiom

\* - something having a meaning that can not be [derived](#) from the conjoined meanings.  
Meaning can be special for language speakers or human with particular knowledge.

\* can also mean [applicative functor](#).

### 29.7.1 \*

Idioms

Idiomatic

## 29.8 Impredicative

Self-referencing definition.

—

*Antonym* - [Predicative](#).

Table 29.1: Ancient Greek and Latin prefixes

Meaning	Greek prefix	Latin prefix
above, excess	hyper-	super-, ultra-
across, beyond, through	dia-	trans-
after		post-
again, back		re-
against	anti-	contra-, (in-, ob-)
all	pan	omni-
around	peri-	circum-
away or from	apo-, ap-	ab- (or de-)
bad, difficult, wrong	dys-	mal-
before	pro-	ante-, pre-
between, among		inter-
both	amphi-	ambi-
completely or very		de-, ob-
down		de-, ob-
four	tetra-	quad-
good	eu-	ben-, bene-
half, partially	hemi-	semi-
in, into	en-	il-, im-, in-, ir-
in front of	pro-	pro-
inside	endo-	intra-
large	macro-	(macro-, from Greek)
many	poly-	multi-
not*	a-, an-	de-, dis-, in-, ob-
on	epi-	
one	mono-	uni-
out of	ek-	ex-, e-
outside	ecto-, exo-	extra-, extro-
over	epi-	ob- (sometimes)
self	auto-, aut-, auth-	ego-
small	micro-	
three	tri-	tri-
through	dia-	trans-
to or toward	epi-	ad-, a-, ac-, as-
two	di-	bi-
under, insufficient	hypo-	sub-
with	sym-, syn-	co-, com-, con-
within, inside	endo-	intra-
without	a-, an-	dis- (sometimes)

## 29.9 Context-free grammar

**Type** of formal grammar that is: a **set** of production rules that describe all possible string is a given formal language.

Term is invented by Noam Chomsky.

### 29.9.1 \*

CFG



# Chapter 30

## Set

Well-defined collection of distinct [objects](#).

### 30.1 \*

Sets  
Set theory

### 30.2 Closed set

- a. [Set](#) which complements an open [set](#).
- b. Is form of [Closed-form expression](#). [Set](#) can be [closed](#) in under a [set](#) of operations.

### 30.3 Power set

For some [set](#)  $\mathcal{S}$ , the [power set](#) ( $\mathcal{P}(\mathcal{S})$ ) is a [set](#) of all subsets of  $\mathcal{S}$ , including  $\{\}$  &  $\mathcal{S}$  itself.

Denotation:  
 $\mathcal{P}(\mathcal{S})$

### 30.4 Singleton

[Singleton](#) - [unit set](#) - [set](#) with exactly one element.  
Also 1-[sequence](#).

### 30.5 Russell's paradox

If there exists normal [set](#) of all [sets](#) - it should contain itself, which makes it abnormal.

## 30.6 Cartesian product

$\mathcal{A} \times \mathcal{B} \equiv \sum^{\forall} (a, b) \mid \forall a \in \mathcal{A}, \forall b \in \mathcal{B}.$

**Operation**, returns a **set** of all ordered pairs  $(a, b)$

Any **function**, **functor** is a subset of **Cartesian product**.

$$\sum (elem \in (\mathcal{A} \times \mathcal{B})) = cardinality^{A \times B}$$

**Properties:**

- not **associative**
- not **commutative**

### 30.6.1 Pullback

Subset of the **cartesian product** of two **sets**.

#### 30.6.1.1 \*

Pullbacks

## Chapter 31

# Shrinking

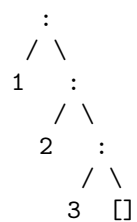
Process of reducing complexity in the test [case](#) - re-run with smaller values and make sure that the test still fails.

## Chapter 32

# Spine

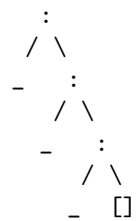
Is a chain of memory cells, each points to the both value of element and to the next memory cell.

Array:



1:2:3: []

Spine:



## Chapter 33

# Superclass

Broader parent class.

## Chapter 34

# Tensor

**Object** existing out of planes, thus it can translate **objects** from one plane into another. They can be tried to be described with knowledge existing inside planes, but representation would always be partial.

**Tensor** of rank 1 is a vector.

Translatioin with **tensor** can be seen as **functors**.

### 34.1 \*

Tensors  
Tensorial

# Chapter 35

## Testing

### 35.1 Property testing

Since [property](#) has a law, then family of that [unit](#) tests can be abstracted into the [lambda function](#).

And tests cases come from [generator](#).

#### 35.1.1 Function property

[Property](#) corresponds to the according law.

In [property testing](#) you need to think additionally about [generator](#) and [shrinking](#).

#### 35.1.2 Property testing type

Table 35.1: [Property testing types](#)

	Exhaustive	Randomized	<a href="#">Unit test</a>
Whole <a href="#">set</a> of values	Exhaustive <a href="#">property test</a>	Randomised <a href="#">property test</a>	One element
Special subset of values	Exhaustive specialised <a href="#">property test</a>	Randomised specialised <a href="#">property test</a>	One element

#### 35.1.3 Generator

Seed  
|  
v  
Gen A -> A  
^  
|  
Size

Seed allows reproducibility.

There is anyway a need to have some seed.

Size allows setting upper [bound](#) on size of generated value. Think about infinity of [list](#).

After failed test - [shrinking](#) tests value parts of contrexample, finds a part that still fails, and recurses [shrinking](#).

**35.1.3.1 \***

Generators

**35.1.3.2 Custom generator**

When certain theorem only works for a specific [set](#) of values - the according [generator](#) needs to be produced.

```
arbitrary :: Arbitrary a => Gen a
suchThat :: Gen a -> (a -> Bool) -> Gen a
elements :: [a] -> Gen a
```

**35.1.4 Reusing test code**

Often it is convinient to [abstract testing](#) of same [function properties](#):

It can be done with (aka TestSuite [combinator](#)):

```
-- Definition
{-# LANGUAGE ScopedTypeVariables #-}
{-# LANGUAGE AllowAmbiguousTypes #-}
eqSpec :: forall a. Arbitrary a => Spec

-- Usage
{-# LANGUAGE TypeApplications #-}
spec :: Spec
spec = do
  eqSpec @Int

Eq Int
(==) :: Int -> Int -> Bool
  is reflexive
  is symetric
  is transitive
  is equivalent to (\ a b -> not $ a /= b)
(/=) :: Int -> Int -> Bool
  is antireflexive
  is equivalent to (\ a b -> not $ a == b)
```

**35.1.4.1 Test Commutative property**

[Commutativity](#)

```
:: Arbitrary a => (a -> a -> a) -> Property
```

**35.1.4.2 Test Symmetry property**

[Symmetry](#)

```
:: Arbitrary a => (a -> a -> Bool) -> Property
```



**35.1.4.3 Test Equivalence property**

Equivalence

```
:: (Arbitrary a, Eq b) => (a -> b) -> (a -> b) -> Property
```

**35.1.4.4 Test Inverse property**

```
:: (Arbitrary a, Eq b) => (a -> b) -> (b -> a) -> Property
```

**35.1.5 QuickCheck**

Target is a member of the [Arbitrary type class](#).

Target -> Bool is something Testable. This [properties](#) can be complex.

[Generator](#) arbitrary gets the seed, and produces values of Target.

[Function](#) quickCheck runs the loop and tests that generated Target values always comply the [property](#).

**35.1.5.1 Manual automation with QuickCheck properties**

```
import Test.QuickCheck
import Test.QuickCheck.Function
import Test.QuickCheck.Property.Common
import Test.QuickCheck.Property.Functor
import Test.QuickCheck.Property.Common.Internal

data Four' a b = Four' a a a b
  deriving (Eq, Show)

instance Functor (Four' a) where
  fmap f (Four' a b c d) = Four' a b c (f d)

instance (Arbitrary a, Arbitrary b) => Arbitrary (Four' a b) where
  arbitrary = do
    a1 <- arbitrary
    a2 <- arbitrary
    a3 <- arbitrary
    b  <- arbitrary
    return (Four' a1 a2 a3 b)

-- Wrapper around 'prop_FunctorId'
prop_AutoFunctorId :: Functor f => f a -> Equal (f a)
prop_AutoFunctorId = prop_FunctorId T

type Prop_AutoFunctorId f a
  = f a
  => Equal (f a)

-- Wrapper around 'prop_AutoFunctorCompose'
prop_AutoFunctorCompose :: Functor f => Fun a1 a2 -> Fun a2 c -> f a1 -> Equal (f c)
prop_AutoFunctorCompose f1 f2 = prop_FunctorCompose (applyFun f1) (applyFun f2) T

type Prop_AutoFunctorCompose structureType origType midType resultType
  = Fun origType midType
  => Fun midType resultType
```

```
structureType origType
Equal (structureType resultType)

main = do
  quickCheck $ eq $ (prop_AutoFunctorId Prop_AutoFunctorId (Four' ()))Integer
  quickCheck $ eq $ (prop_AutoFunctorId Prop_AutoFunctorId (Four' ())) (Either Bool String)
  quickCheck $ eq $ (prop_AutoFunctorCompose Prop_AutoFunctorCompose (Four' ())) String Integer S
  quickCheck $ eq $ (prop_AutoFunctorCompose Prop_AutoFunctorCompose (Four' ())) Integer String (
```

## 35.2 Write tests algorithm

- a. Pick the right language/[stack](#) to implement features.
- b. How expensive breakage can be.
- c. Pick the right tools to test this.

## Chapter 36

# Uncurry

Replace sequenced lambda [functions](#) into single [function](#) taking [sequence/product](#) of values as [argument](#).

## Chapter 37

# Unit

Represents existence. Denoted as empty [sequence](#).

()

Type () holds only self-representation [constructor](#) (), & [constructor](#) holds [nothing](#).

Haskell code always should receive something back, hence [nothing](#), emptiness, [void](#) can not be theoretically addressed, practically constructed or received - [unit](#) in Haskell also has a role of a stub in place of emptiness, like in IO ().

## Chapter 38

# Nullary

Takes no entries; has the [arity](#) of [zero](#).  
Has the trivial [domain](#).

# Chapter 39

## Logic

### 39.1 Proposition

Purely abstract & theoretical logical [object](#) (idea) that has a Boolean value.

\* is expressed by a [statement](#).

#### 39.1.1 \*

Propositions

#### 39.1.2 Atomic proposition

Logically undividable [unit](#). Does not contain [logical connectives](#).

##### 39.1.2.1 \*

Atomic propositions

#### 39.1.3 Compound proposition

Formed by connecting [propositions](#) by [logical connectives](#).

##### 39.1.3.1 \*

Compound propositions

#### 39.1.4 Propositional logic

Studies [propositions](#) and [argument](#) flow.

Refers to logically indivisible units ([atomic propositions](#)) as such, for theory - they are [abstractions](#) with Boolean [properties](#).

Not Turing-complete, impossible to [construct](#) an [arbitrary](#) loop.

**39.1.4.1 \***

Proposition logic  
 Proposition calculus  
 Propositional calculus  
 Statement logic  
 Sentential logic  
 Sentential calculus  
 Zeroth-order logic

**39.1.4.2 First-order logic**

Notation systems that use [quantifiers](#), [relations](#), [variables](#) over non-logical [objects](#), allows the use of [expressions](#) that contain [variables](#).

Turing-complete.

Extension of a [propositional logic](#).

*a.* \*

Predicate logic  
 First-order predicate logic  
 First-order predicate calculus

*b.* Second-order logic Extension over [first-order logic](#) that quantifies over [relations](#).*a.* Higher-order logic Extension over [second-order logic](#) that uses additional [quantifiers](#), stronger [semantics](#).

Is more expressive, but model-theoretic [properties](#) are less well-behaved.

**39.2 Logical connective**

Logical [operation](#).

**39.2.1 \***

Logical connectives

**39.2.2 Conjunction**

Logical AND.

Denotation:

$\wedge$

Multiplies [cardinalities](#).

Haskell [kind](#):

\* \*

### 39.2.3 Disjunction

Logical *OR*

Denotation:

$\vee$

Summs [cardinalities](#).

## 39.3 Predicate

[Function](#) with Boolean [codomain](#).

$P : X \rightarrow \{true, false\}$  - \* on  $X$ .

Notation:  $P(x)$

Almost always can include [relations](#), [quantifiers](#).

## 39.4 Statement

Declarative [expression](#) that is a bearer of a [proposition](#).

When we talk about [expression](#) or [statement](#) being true/false - in fact we refer to the [proposition](#) that they represent.

Difference between [proposition](#), [statement](#), [expression](#):

a. "2 + 3 = 5"

b. "two plus three equals five"

- 1 & 2 are [statements](#). Each of them is a collection of transmission symbols (linguistic [objects](#)) from a symbol systems  $\equiv$  [expression](#). Each of them is [expression](#) that bears [proposition](#) (an idea resulting in a Boolean value)  $\equiv$  [statement](#).
- 1 & 2 represent the same [proposition](#). [Proposition](#) from 1  $\equiv$  [proposition](#) from 2.
- [Statement](#) 1  $\neq$  [statement](#) 2. They are two different [statements](#), written in different systems. And [statement](#) "2 + 3 = 5"  $\neq$  [statement](#) "3 + 2 = 5".

### 39.4.1 \*

Assertion

Assertions

Statements



## 39.5 Iff

If and only if, exactly when, just.

Denotation:

$\iff$

# Chapter 40

## Haskell structures

### 40.1 Pattern match

Are not [first-class](#). It is a [set](#) of patten match semantic notations.

Must be [linear](#).

\* [precedence](#) (especially with more then one [parameter](#), especially with `_` used) often changes the [function](#).

#### 40.1.1 As-pattern

```
f list@(x, xs) = ...
```

```
f (x:xs)    = x:x:xs -- Can be compiled with reconstruction of x:xs
f a@(x:_)   = x:a    -- Reuses structure without reconstruction
```

##### 40.1.1.1 \*

As-patterns  
As pattern  
As patterns

#### 40.1.2 Wild-card

Matches anything and can not be binded. For matching someting that should pass not checked and is not used.

```
head (x:_)      = x
tail (_:xs)     = xs
```

##### 40.1.2.1 \*

Wild-cards  
Wildcard  
Wildcards

### 40.1.3 Case

```
case x of
  pattern1 -> ex1
  pattern2 -> ex2
  pattern3 -> ex3
  otherwise -> exDefault
```

Bolting [guards](#) & [expressions](#) with [syntactic sugar](#) on [case](#):

```
case () of _
  | expr1      -> ex1
  | expr2      -> ex2
  | expr3      -> ex3
  | otherwise  -> exDefault
```

Pattern matching in [function](#) definitions is realized with [case expressions](#).

### 40.1.4 Guard

Check values against the [predicate](#) and use the first match definition:

```
f x
  | predicate1 = definition1
  | predicate2 = definition2
  ...
  | x < 0      = definitionN
  ...
  | otherwise  = definitionZ
```

#### 40.1.4.1 \*

Guards

### 40.1.5 Pattern guard

Allows check a [list](#) of pattern matches against [functions](#), and then proceed.

$$(\text{patternMatch1}) <- (\text{funcCheck1})$$

```
, (\text{patternMatch2}) <- (\text{funcCheck2})
= RHS
```

```
lookup :: Eq a => a -> [(a, b)] -> Maybe b
```

```
addLookup l a1 a2
  | Just b1 <- lookup a1 l
  , Just b2 <- lookup a2 l
  = b1 + b2
{-...other equations...-}
```

Run [functions](#), they must succeed. Then [pattern match](#) results to `b1`, `b2`. Only if successful - execute the equation.

Default in Haskell 2010.

#### 40.1.5.1 \*

Pattern guards

### 40.1.6 Lazy pattern

Defers the [pattern match](#) directly to the last moment of need during execution of the code.

```
f (a, b) = g a b -- It would be checked that the pattern of the pair constructor
-- is present, and that parameters are present in the constructor.
-- Only after that success - work would start on the RHS, aka then construction
-- g would start only then.
```

```
f ~(a, b) = g a b -- Pattern match of (a, b) deferred to the last moment,
-- RHS starts, construction of g starts.
-- For this lazy pattern the equivalent implementation would be:
-- f p = g (fst p) (snd p) -- RHS starts, during construction of g
-- the arguments would be computed and found, or error would be thrown.
```

Due to full laziness deferring everything to the runtime execution - the [lazy pattern](#) is one-size-fits all ([irrefutable](#)), analogous to `_`, and so it does not produce any checks during compilation, and raises [errors](#) during runtime.

`*` is very useful during [recursive](#) construction of [recursive structure](#)/process, especially infinite.

#### 40.1.6.1 \*

Lazy-pattern  
Lazy patterns

### 40.1.7 Pattern binding

Entire [LHS](#) is a pattern, is a [lazy pattern](#).

```
fib@1:tfib)    = 1 : 1 : [ a+b | (a,b) <- zip fib tfib ]
```

#### 40.1.7.1 \*

Pattern bindings

## 40.2 Smart constructor

Process/code placing extra rules & [constraints](#) on the construction of values.

## 40.3 Level of code

There are these levels of Haskell code:

### 40.3.1 \*

Code level

### 40.3.2 Type level

[Level of code](#) that works with [data types](#).

#### 40.3.2.1 Type level declaration

```
type ...
newtype ...
data ...
class ...
instance ...
```

*a.* \*

Type level declarations  
Type-level declaration  
Type-level declarations

#### 40.3.2.2 Type check

if The [type level](#) information is complete ([strongly connected](#) graph)

then

Generalize the [types](#) and check if [type level](#) consistent to [term level](#).

else

[Infer](#) the missing [type level](#) part from the [term level](#). There are certain situations and [structures](#) [where](#) ambiguity arises and is unsolvable from the information of the [term level](#) (most basic example is [polymorphic recursion](#)).

*a.* \*

Typecheck  
Typechecking  
Typechecks

- b.* Complete user-specific kind signature [Type level declaration](#) is considered to "have a [CUSK](#)" is it has enough syntatic information to warrant completeness ([strongly connected](#) graph) and start checking [type level](#) correspondence to [term level](#), it is a ad-hock state of [type inferring](#).

In the future GHC would use other algorythm over/instead of [CUSK](#).

- a.* \*
  - CUSK
  - CUSKs
  - Complete user-specific kind signatures
  - Complete, user-specific kind signature

### 40.3.3 Term level

[Level of code](#) that does logical execution.

### 40.3.4 Compile level

[Level of code](#), about compilation processes/results.

#### 40.3.4.1 \*

Compilation level

### 40.3.5 Runtime level

[Level of code](#) of main program [operation](#), when machine does computations with compiled [binary](#) code.

### 40.3.6 Kind level

[Level of code](#) where [kinds](#) & [kind](#) declarations are situated, inferred and checked.

#### 40.3.6.1 Kind check

Applying the [type check](#) to [kind](#) check:

if The [kind](#) level information is complete ([strongly connected](#) graph)

then

Check if [kind](#) level consistent to [term level](#).

else

Infer the missing [kind](#) level parts from the [type level](#). There are certain situations and [structures](#) where ambiguity arises and is unsolvable from the information of the [kind](#) level.

With `StandaloneKindSignatures` [kind](#) completeness happens against found (standalone) [kind](#) signature.

With `CUSKs` extension kind completeness happens against "[complete user-specific kind signature](#)"

- a. \*  
Kindcheck  
Kind checks

## 40.4 Orphan type instance

Hanging [type](#) instance from inconsistent code base.

- a. Supporting [structure](#) not fully present.
- b. Several implementations of instance present.

## 40.5 Undefined

Placeholder value that helps to do [typechecking](#).

## 40.6 Hierarchical module name

Hierarchical naming scheme:

```

Algebra                                -- Was this ever used?
  DomainConstructor                    -- formerly DoCon
  Geometric                            -- formerly BasGeomAlg

Codec                                  -- Coders/Decoders for various data formats
  Audio
    Wav
    MP3
    ...
  Compression
    Gzip
    Bzip2
    ...
  Encryption
    DES
    RSA
    BlowFish
    ...
  Image
    GIF
    PNG
    JPEG
    TIFF
    ...
  Text
    UTF8
    UTF16
    ISO8859

```

```

    ...
Video
  Mpeg
  QuickTime
  Avi
  ...
Binary          -- these are for encoding binary data into text
  Base64
  Yenc

Control
  Applicative
  Arrow
  Exception      -- (opt, inc. error & undefined)
  Concurrent      -- as hslibs/concurrent
    Chan          -- these could all be moved under Data
    MVar
    Merge
    QSem
    QSemN
    SampleVar
    Semaphore
  Parallel        -- as hslibs/concurrent/Parallel
  Strategies
  Monad           -- Haskell 98 Monad library
    ST            -- ST defaults to Strict variant?
      Strict      -- renaming for ST
      Lazy        -- renaming for LazyST
    State         -- defaults to Lazy
      Strict
      Lazy
    Error
    Identity
    Monoid
    Reader
    Writer
    Cont
    Fix           -- to be renamed to Rec?
    List
    RWS

Data
  Binary          -- Binary I/O
  Bits
  Bool            -- &&, ||, not, otherwise
  Tuple          -- fst, snd
  Char           -- H98
  Complex        -- H98
  Dynamic
  Either
  Int
  Maybe          -- H98
  List           -- H98
  PackedString
  Ratio          -- H98

```



```

Word
IORef
STRef          -- Same as Data.STRef.Strict
    Strict
    Lazy        -- The lazy version (for Control.Monad.ST.Lazy)
Binary          -- Haskell binary I/O
Digest
    MD5
    ...         -- others (CRC ?)
Array           -- Haskell 98 Array library
    Unboxed
    IArray
    MArray
    IO          -- mutable arrays in the IO/ST monads
    ST
Trees
    AVL
    RedBlack
    BTree
Queue
    Bankers
    FIFO
Collection
Graph           -- start with GHC's DiGraph?
FiniteMap
Set
Memo            -- (opt)
Unique

Edison          -- (opt, uses multi-param type classes)
    Prelude     -- large self-contained packages should have
    Collection  -- their own hierarchy? Like a vendor branch.
    Queue       -- Or should the whole Edison tree be placed

Database
    MySQL
    PostgreSQL
    ODBC

Dotnet
    ...         -- Mirrors the MS .NET class hierarchy

Debug           -- see also: Test
    Trace
    Observe     -- choose a default amongst the variants
        Textual      -- Andy Gill's release 1
        ToXmlFile    -- Andy Gill's XML browser variant
        GHood        -- Claus Reinke's animated variant

Foreign
    Ptr
    StablePtr
    ForeignPtr  -- rename to FinalisedPtr? to void confusion with Foreign.Ptr
    Storable
    Marshal

```

```
    Alloc
    Array
    Errors
    Utils
C
    Types
    Errors
    Strings

GHC
    Exts          -- hslibs/lang/GlaExts
    ...

Graphics
    HGL
    Rendering
        Direct3D
        FRAN
        Metapost
        Inventor
        Haven
        OpenGL
            GL
            GLU
    Pan
UI
    FranTk
    Fudgets
    GLUT
    Gtk
    Motif
    ObjectIO
    TkHaskell
X11
    Xt
    Xlib
    Xmu
    Xaw

Hugs
    ...

Language
    Haskell      -- hslibs/hssource
        Syntax
        Lexer
        Parser
        Pretty
    HaskellCore
    Python
    C
    ...

Nhc
    ...
```

```

Numeric          -- exports std. H98 numeric type classes
  Statistics

Network          -- (== hslibs/net/Socket), depends on FFI only
  BER            -- Basic Encoding Rules
  Socket         -- or rename to Posix?
  URI            -- general URI parsing
  CGI            -- one in hslibs is ok?
  Protocol
    HTTP
    FTP
    SMTP

Prelude          -- Haskell98 Prelude (mostly just re-exports
                  other parts of the tree).

Sound            -- Sound, Music, Digital Signal Processing
  ALSA
  JACK
  MIDI
  OpenAL
  SC3            -- SuperCollider

System           -- Interaction with the "system"
  Cmd            -- ( system )
  CPUTime        -- H98
  Directory      -- H98
  Exit           -- ( ExitCode(..), exitWith, exitFailure )
  Environment    -- ( getArgs, getProgName, getEnv ... )
  Info           -- info about the host system
  IO             -- H98 + IOExts - IOArray - IORef
    Select
    Unsafe       -- unsafePerformIO, unsafeInterleaveIO
  Console
    GetOpt
    Readline
  Locale         -- H98
  Posix
    Console
    Directory
    DynamicLinker
      Prim
      Module
    IO
    Process
    Time
  Mem            -- rename from cryptic 'GC'
    Weak         -- (opt)
    StableName   -- (opt)
  Time          -- H98 + extensions
  Win32         -- the full win32 operating system API

Test
  HUnit

```

```

QuickCheck

Text
  Encoding
    QuotedPrintable
    Rot13
  Read
    Lex          -- cut down lexer for "read"
  Show
    Functions    -- optional instance of Show for functions.
  Regex         -- previously RegexString
  Posix         -- Posix regular expression interface
  PrettyPrint   -- default (HughesPJ?)
    HughesPJ
    Wadler
    Chitil
    ...
  HTML          -- HTML combinator lib
  XML
    Combinators
    Parse
    Pretty
    Types
  ParserCombinators -- no default
    ReadP       -- a more efficient "ReadS"
    Parsec
    Hutton_Meijer
    ...

Training        -- Collect study and learning materials
  <name of the tutor>

```

### 40.6.1 \*

Top-level module name  
 Top-level module names

## 40.7 import

`import` [statement](#) by default imports identifiers from the other [module](#), using [hierarchical module name](#), brings into [scope](#) the identifiers to the global [scope](#) both into unqualified and qualifies by the [hierarchical module name](#) forms.

This possibilities can mix and match:

- `<modName> ()` - `import` only instances of [type classes](#).
- `<modName> (x, y)` - `import` only declared indentifiers.
- `qualified <modName>` - discards unqualified names, forse obligatory namespace for the imports.

- `hiding (x, y)` - skip `import` of declared identifies.
- `<modName> as <modName>` - renames `module` namespace.
- `<type/class> (..)` - `import` class & it's methods, or `type`, all its data `constructors` & field names.

## 40.8 Let

\* `expression` is a `set` of cross-recursive lazy pattern bindings.

Declarations permitted:

- `type` signatures
- `function bindings`
- `pattern bindings`

It is an `expression` (macro) and that integrates in external `lexical scope expression` it `applied` in.

Form:

```
let
  b1
  bn
in
  c
```

### 40.8.1 \*

Let expression  
Let expressions

## 40.9 Where

Part of the syntax of the whole `function declaration`, has according `scope`.

As part of whole `declaration` - can extend over definitions of the funtion (pattern matches, `guards`).

Form:

```
f match1 = y
f match2 = y
f x =
  | cond1 x = y
```

```

    | cond2 x = y
    | otherwise = y
where
    y = ... x ...

```

### 40.9.1 \*

Where clause

## 40.10 Reserved word

Haskell has special meaning for:

case, class, data, deriving, do, else, if, import, in, infix, infixl, infixr, instance, let, of, module

### 40.10.1 \*

Reserved words

## 40.11 Haskell Language Report

Document that is a standard of language.

### 40.11.1 \*

Report  
Haskell Report  
Haskell 98 Language Report  
Haskell 98 Report  
Haskell 1998 Language Report  
Haskell 2010 Language Report  
Haskell 2010 Report

## 40.12 Haskell'

Current language development mod.

<https://prime.haskell.org/>

### 40.12.1 \*

Haskell prime

# Chapter 41

## Computer science

### 41.1 Guerrilla patch

\* changing code/[applying](#) patch sneakily - and possibility incompatibility with other at runtime. [Monkey patch](#) is derivative term.

#### 41.1.1 Monkey patch

From [Guerrilla patch](#).

\* is a way for program to modify supporting system software affecting only the running instance of the program.

### 41.2 Interface

Point of mutual meeting. Code behind [interface](#) determines how data is consumed.

### 41.3 Module

Importable organizational [unit](#).

### 41.4 Scope

Area [where binds](#) are accessible.

#### 41.4.1 Dynamic scope

The name resolution depends upon the program state when the name is encountered, which is determined by the execution [context](#) or calling [context](#).

#### 41.4.2 Lexical scope

[Scope bound](#) by the [structure](#) of source code [where](#) the named entity is defined.

**41.4.2.1 \***

Static scope

**41.4.3 Local scope**

Scope applies only in (current) area.

**41.4.3.1 \***

Local

**41.5 Shadowing**

When in the local scope bigger scope variable overridden by same name variable from the local scope.

**41.6 Syntactic sugar**

Artificial way to make language easier to read and write.

**41.7 System F**

Is parametric polymorphism in programming.

Extends the Lambda calculus by introducing  $\forall$  (universal quantifier) over types.

**41.7.1 \***

Girard–Reynolds polymorphic lambda calculus  
Girard-Raynolds

**41.8 Tail call**

Final evaluation inside the function. Produces the function result.

**41.9 Thunk**

Not evaluated calculation. Can be dragged around, until be lazily evaluated.



Table 41.1: Application memory structural parts

Storage of	Block name
All not currently processing data	Heap
Function call, local variables	Stack
Static and global variables	Static/Global
Instructions	Binary code

## 41.10 Application memory

When even Main invoked - it work in [Stack](#), and called [Stack](#) frame. [Stack](#) frame size for [function](#) calculated when it is compiled.

When stacked [Stack](#) frames exceed the [Stack](#) size - [stack](#) overflow happens.

## 41.11 Turing machine

Mathematical model of computation that defines [abstract Turing machine](#). [Abstract](#) machine which manipulates symbols on a strip of tape, according to a table of rules.

### 41.11.1 Turing complete

[Set](#) of action rules that can simulate any [Turing machine](#).

#### 41.11.1.1 \*

Turing incomplete

Turing incompleteness

Turing completeness

Computationally universal

# Chapter 42

## Graph theory

### 42.1 Successor

**Object** that receives the **arrow**.

#### 42.1.1 Direct successor

Immediate **successor**.

### 42.2 Predecessor

**Object** that sends **arrow**.

#### 42.2.1 Direct predecessor

Immediate **predecessor**.

### 42.3 Degree

Number of **arrows** of **object**.

#### 42.3.1 Indegree

Number of ingoing **arrows**.

#### 42.3.2 Outdegree

Number of outgoing **arrows**.

### 42.4 Adjacency matrix

Matrix of connection of objects  $\{-1, 0, 1\}$ .

**42.4.0.1 InstanceSigs**

Allow adding [type](#) signatures to [type class function](#) instance [declaration](#).

**42.5 Strongly connected**

If every vertex in a graph is reachable from every other vertex.

It is possible to find all [strongly connected components](#) (and that way also test graph for strong connectivity), in [linear](#) time  $(V+E)$ .

[Binary relation](#) of being [strongly connected](#) is an [equivalence relation](#).

**42.5.1 \***

Strongly-connected

**42.5.2 Strongly connected component**

Full [strongly connected](#) subgraph of some graph.

\* of a directed graph  $G$  is a subgraph that is [strongly connected](#), and has [property](#): no additional edges or vertices from  $G$  can be included in the subgraph without breaking its [property](#) of being [strongly connected](#).

**42.5.2.1 \***

SCC

Strongly connected components

Strongly-connected component

Strongly-connected components

## Chapter 43

# Linear

Values consumed once or not used.

$x^2$  consumes  $x$  two times.

### 43.1 \*

Linearity

## Chapter 44

# Stream

\* an infinite [sequence](#) that forgets previous [objects](#), and remembers only currently relevant [objects](#).

$E \mid X \rightarrow (X \times A + 1)$ , the [set](#) (or [object](#)) of streams on  $A$  (final [coalgebra](#)  $A_*$  of  $E$ ).

`cycle` is one of [stream functions](#).

```
a = (cycle [Nothing, Nothing, Just "Fizz"])
b = (cycle [Nothing, Nothing, Nothing, Nothing, Just "Buzz"])
```

Can be:

- indexed, timeless, with current [object](#)
- timed:

```
* [(timescale, event)]
* [(realtime, event)]
```

Has amalgamation with Functional Reactive Programming.

## Chapter 45

# Bisimulation

When systems have exact external behaviour so for observer they are the same.

[Binary relation](#) between state transition systems that match each other's moves.

### 45.1 \*

Bisimilar

# Chapter 46

## Syntax tree

Tree of syntactic elements (each **node** denotes **construct** occurring in the source code) that represent the source code (or human language).

### 46.1 Abstract syntax tree

"**Abstract**" since does not represent every detail of the syntax (ex. parentheses), but rather concentrates on **structure** and content.

Widely used in compilers to check the code **structure** for accuracy and coherence.

#### 46.1.1 \*

AST

### 46.2 Concrete syntax tree

An ordered, rooted **syntax tree** that represents the syntactic **structure** of a string according to some **context-free grammar**.

"Concrete" since (in contrast to "**abstract**") - concretely reflects the syntax of the input language.

#### 46.2.1 \*

Parse tree

Derivation tree

## Chapter 47

# Domain specific language

Language design/fitted for particular [domain](#) of [application](#). Mainly should be [Turing incomplete](#), since general-purpose language implies [Turing completeness](#).

### 47.1 \*

Domain-specific language  
DSL

### 47.2 Embedded domain specific language

[DSL](#) used inside outer language.

Two levels of embedding:

- Shallow: [DSL](#) translates into Haskell directly
- Deep: Between [DSL](#) and Haskell there is a [data structure](#) that reflects the [expression tree](#), AKA stores the [syntax tree](#).

#### 47.2.1 \*

eDSL



## Chapter 48

# Tagless-final

Method of embedding **eDSL** in a typed functional host language (Haskell). **Alternative** to the embedding as a (generalized) **algebraic data type**. For parsers of DLS **expressions**: (1/partial) evaluator, compiler, pretty printer, multi-pass optimizer.

\* embedding is writing **denotational semantics** for the **DSL** in the host language.

Approach can be used **iff eDSL** is typed. Only well-typed terms become embeddable, and host language can implement also a **eDSL type** system. Approach that **eDSL** code interpretations are **type-preserving**.

One of main pros of \* - extensibility: implementation of **DSL** can be used to analyze/evaluate/transform/pretty-print/compile and interpreters can be extended to more passes, optimizations, and new versions of **DSL** while keeping/using/reusing the old versions.

Example fields of **application**: language-integrated queries, non-deterministic & probabilistic programming, delimiter continuation, computability theory, **stream** processing, hardware description languages, generation of specialized numerical kernels, **semantics** of natural language.

## Part III

# Give definitions

## Chapter 49

### Identity type

## Chapter 50

### Constant type

## Chapter 51

## Gen

## Chapter 52

# Tensorial strength

## Chapter 53

# Strong monad

## Chapter 54

# Weak head normal form

### 54.1 \* WHNF



## Chapter 55

# Function image

### 55.1 \*

Image

## Chapter 56

# Invertible

## Chapter 57

# Invertibility

## Chapter 58

# Define LANGUAGE pragma options

### 58.1 ExistentialQuantification

### 58.2 GADTs

**GADT** is a generalization over parametric [algebraic data types](#) which allow explicitly denote the [types](#) ([type matching](#)) of the [constructors](#) and define [data types](#) using pattern matching on the left side of "data" [statements](#).

### 58.3 \*

GADT

Generalized algebraic data type

First-class phantom data type

Guarded recursive data type

Equality-qualified data type

### 58.4 GeneralizedNewTypeClasses

### 58.5 FuncitonalDependencies

## Chapter 59

# GHC check keys

### 59.1 -Wno-partial-type-signatures

Supresses [PartialTypeSignatures wildcard infer](#) warning.

## Chapter 60

# Generalised algebraic data types

LANGUAGE [GADTs](#)

### 60.1 \*

GADT

# Chapter 61

## Order theory

Investigates in thepeth the intuitive notion of [order](#) using [binary relations](#).

### 61.1 Domain theory

Formalizes approximation and convergense.  
Has close [relation](#) to Topology.

### 61.2 Lattice

[Abstract structure](#) that consists of [partially ordered set](#), [where](#) every two elements have unique supremum and infimum. == \* [algebraic structure](#) satisfying certain axiomatic identities.  
\* [order-theory](#) & [algebraic](#).

### 61.3 Order

#### 61.3.1 Preorder

$R^X \rightarrow X : \text{Reflexive} \ \& \ \text{Transitive}:$   
 $aRa$   
 $aRb, bRc \Rightarrow aRc$

Generalization of [equivalence relations](#) [partial orders](#).

\* [Antisymmetric](#)  $\Rightarrow$  Partial ordering.  
\* [Symmetric](#)  $\Rightarrow$  [Equivalence](#).

##### 61.3.1.1 \*

Preordered

##### 61.3.1.2 Total preorder

$\forall a, b : a \leq b \vee b \leq a \Rightarrow \text{Total Preorder}.$

### 61.3.2 Partial order

A [binary relation](#) must be [reflexive](#), [antisymmetric](#) and [transitive](#).

Partial - not every elements between them need to be comparable.

Good example of  $*$  is a genealogical descendancy. Only related people produce [relation](#), not related do not.

#### 61.3.2.1 $*$

Partial orders

Partially ordered set

Partially ordered sets

Poset

Posets

## 61.4 Partial order

## 61.5 Total order



## Chapter 62

# Universal algebra

Studies [algebraic structures](#).

# Chapter 63

## Relation

### 63.1 Reflexivity

$$R^{X \rightarrow X}, \forall x \in X : xRx$$

[Order](#) theory:  $a \leq a$

\* - each element is comparable to itself.

Corresponds to [Identity](#) and [Automorphism](#).

#### 63.1.1 \*

Reflexive

Reflexive relation

### 63.2 Irreflexivity

$$R^{X \rightarrow X}, \forall x \in X : \nexists R(x, x)$$

#### 63.2.1 \*

Anti-reflexive

Anti-reflexive relation

Irreflexive

Irreflexive relation

### 63.3 Transitivity

$$\forall a, b, c \in X, \forall R^{X \rightarrow X} : (aRb \wedge bRc) \Rightarrow aRc$$

\* - the start of a chain of [precedence relations](#) must precede the end of the chain.

**63.3.1 \***

Transitive  
Transitive relation

**63.4 Symmetry**

$$\forall a, b \in X : (aRb \iff bRa)$$

**63.4.1 \***

Symmetric  
Symmetric relation

**63.5 Equivalence**

Reflexive	Symmetric	Transitive
$\forall x \in X, \exists R : xRx$ $a = a$	$\forall a, b \in X : (aRb \iff bRa)$ $a = b \iff b = a$	$\forall a, b, c \in X, \forall R^{X \rightarrow X} : (aRb \wedge bRc) \Rightarrow aRc$ $a = b, b = c \Rightarrow a = c$

**63.5.1 \***

Equivalent  
Equivalent relation

**63.6 Antisymmetry**

$$\forall a, b \in X : aRb, bRa \Rightarrow a = b \sim aRb, a \neq b \Rightarrow \nexists bRa.$$

[Antisymmetry](#) does not say anything about  $R(a, a)$ .

\* - no two different elements precede each other.

**63.6.1 \***

Antisymmetric  
Antisymmetric relation

**63.7 Asymmetry**

$$\forall a, b \in X (aRb \Rightarrow \neg(bRa))$$

\*  $\iff$  [Antisymmetric](#)  $\wedge$  [Irreflexive](#).

[Asymmetry](#)  $\neq$  "not [symmetric](#)"

[Symmetric](#)  $\wedge$  [Asymmetric](#) is only empty relation.

**63.7.1   \***

Asymmetric

Asymmetric relation

## Chapter 64

# Cryptomorphism

[Equivalent](#), interconvertable with no loss of information.

### 64.1 \*

Crypromorphic

## Chapter 65

# Lexically scoped type variables

Enable [lexical scope](#) for [forall quantifier](#) defined [type variables](#)

Implemented in [ScopedTypeVariables](#)

## Chapter 66

# Abstract data type

Several definitions here, reduce them.

**Data type** mathematical model, defined by its **semantics** from the user point of view, listing possible values, operations on the data of the **type**, and behaviour of these operations.

\* class of **objects** whose logical behaviour is defined by a **set** of values and **set** of operations (analogue to **algebraic structure** in mathematics).

A specification of a **data type** like a **stack** or queue **where** the specification does not contain any implementation details at all, only the operations for that **data type**. This can be thought of as the contract of the **data type**.

### 66.1 \*

AbsDT

## Chapter 67

# Concrete type

Fully defined [type](#). Non-[polymorphic type](#).



## Chapter 68

# Functional dependencies

## Chapter 69

# MonoLocalBinds

## Chapter 70

# KindSignatures

## Chapter 71

# ExplicitNamespaces

## Chapter 72

# Combinator pattern

## Chapter 73

# Symbolic expression

Nested tree [data structure](#).

Introduced & used in Lisp. Lisp code and data are  $*$ .

$*$  in Lisp: Atom or [expression](#) of the form  $(x \ . \ y)$ ,  $x$  and  $y$  are  $*$ .

Modern abbreviated notation of  $*$ :  $(x \ y)$ .

### 73.1 $*$

S-expression

S-expressions

Sexpression

Sexpressions

Sexp

Sexps

Sexpr

Sexprs

## Chapter 74

# Polynomial

[Expression](#) consisting of:

- [variables](#)
- coefficients
- addition
- subtraction
- multiplication (including positive integer [variable](#) exponentiation)

[Polynomials](#) form a [ring](#). [Polynomial ring](#).

### 74.1 \*

Polynomials

## Chapter 75

# Data family

Indexed form of data and newtype definitions.



## Chapter 76

# Type synonym family

Indexed form of [type](#) synonyms.

## Chapter 77

# Indexed type family

\* additional structure in language that allows ad-hoc overloading of [data types](#). AKA are to [types](#) as [type class](#) to methods.

Varieties:

- [data family](#)
- [type](#) synonym families

Defined by pattern matching the partial [functions](#) between [types](#).

Associates [data types](#) by [type](#)-level [function](#) defined by open-ended collection of valid instances of input [types](#) and corresponding output [types](#).

Normal [type classes](#) define partial [functions](#) from [types](#) to a collection of named values by pattern matching on the input [types](#), while [type](#) families define partial [functions](#) from [types](#) to [types](#) by pattern matching on the input [types](#). In fact, in many uses of [type](#) families there is a single [type class](#) which logically contains both values and [types](#) associated with each instance. A [type family](#) declared inside a [type class](#) is called an associated [type](#).

### 77.1 \*

Type family

## Chapter 78

# TypeFamilies

Allow use and definition of indexed [type](#) families and data families.

- \* are [type](#)-level programming.
- \* are overload [data types](#) in the same way that [type classes](#) overload [functions](#).
- \* allow handling of [dependent types](#). Before it [Functional dependencies](#) and [GADTs](#) were used to solve that.
- \* useful for generic programming, creating highly parametrised interfaces for libraries, and creating interfaces with enhanced static information (much like [dependent types](#)).

Implies: [MonoLocalBinds](#), [KindSignatures](#), [ExplicitNamespaces](#)

Two [types](#) of \* are:

## Chapter 79

# Error

Mistake in the program that can be resolved only by fixing the program.

`error` is a sugar for `undefined`.

Distinct from [Exception](#).

### 79.1 \*

Errors

## Chapter 80

# Exception

Expected but irregular situation.

Distinct from [Error](#). Also see Exception vs Error

### 80.1 \*

Exceptions

## Chapter 81

# ConstraintKinds

[Constraints](#) are just handled as [types](#) of a particular [kind](#) ([Constraint](#)). Any [type](#) of the [kind](#) [Constraints](#) can be used as a [constraint](#).

- Anything which is already allowed in code as a [constraint](#) without `*`. Saturated applications to [type classes](#), implicit [parameter](#) and equality [constraints](#).
- [Tuples](#), all of whose component [types](#) have [kind](#) [Constraint](#).

```
type Some a = (Show a, Ord a, Arbitrary a) -- is of kind Constraint.
```

- Anything form of which is not yet known, but the user has declared for it to have [kind](#) [Constraint](#) (for which they need to [import](#) it from `GHC.Exts`):

```
Foo (f :: Type -> Constraint) = forall b. f b => b -> b -- is allowed
-- as well as examples involving type families:
type family Typ a b :: Constraint
type instance Typ Int b = Show b
type instance Typ Bool b = Num b

func :: Typ a b => a -> b -> b
func = ...
```

## Chapter 82

# Specialisation

Turns [ad hoc polymorphic function](#) into compiled [type](#)-specific implementations.

### 82.1 \*

Specialise  
Specialize  
Specialization

## Chapter 83

# Diagram

For [categories](#)  $\mathcal{C}$  and  $\mathcal{J}$ , a [diagram](#) of [type](#)  $\mathcal{J}$  in  $\mathcal{C}$  is a [covariant functor](#)  $D : \mathcal{J} \rightarrow \mathcal{C}$ .



## Chapter 84

# Category theoretical presheaf

For [categories](#)  $C$  and  $J$ , a  $J$ -[presheaf](#) on  $C$  is a [contravariant functor](#)  $D : C \rightarrow J$ .

## Chapter 85

# Topological presheaf

If  $X$  is a topological space, then the open sets in  $X$  form a [partially ordered set](#)  $\text{Open}(X)$  under inclusion. Like every [partially ordered set](#),  $\text{Open}(X)$  forms a small [category](#) by adding a single [arrow](#)  $U \rightarrow V$  if and only if  $U \subseteq V$ . [Contravariant functors](#) on  $\text{Open}(X)$  are called presheaves on  $X$ . For instance, by assigning to every open set  $U$  the [associative algebra](#) of real-valued continuous functions on  $U$ , one obtains a [presheaf](#) of [algebras](#) on  $X$ .

## Chapter 86

# Diagonal functor

The [diagonal functor](#) is defined as the [functor](#) from  $D$  to the [functor category](#)  $D^C$  which sends each [object](#) in  $D$  to the [constant functor](#) at that [object](#).

## Chapter 87

# Limit functor

For a fixed index [category](#)  $J$ , if every [functor](#)  $J \rightarrow C$  has a limit (for instance if  $C$  is complete), then the [limit functor](#)  $C^J \rightarrow C$  assigns to each [functor](#) its limit. The existence of this [functor](#) can be proved by realizing that it is the right-adjoint to the [diagonal functor](#) and invoking the Freyd adjoint [functor](#) theorem. This requires a suitable version of the axiom of choice. Similar remarks [apply](#) to the colimit [functor](#) (which is [covariant](#)).

## Chapter 88

# Dual vector space

The map which assigns to every vector space its [dual](#) space and to every [linear](#) map its [dual](#) or transpose is a [contravariant functor](#) from the [category](#) of all vector spaces over a fixed field to itself.

## Chapter 89

# Fundamental group

Consider the [category](#) of pointed topological spaces, i.e. topological spaces with distinguished points. The [objects](#) are pairs  $(X, x_0)$ , [where](#)  $X$  is a topological space and  $x_0$  is a point in  $X$ . A [morphism](#) from  $(X, x_0)$  to  $(Y, y_0)$  is given by a continuous map  $f : X \rightarrow Y$  with  $f(x_0) = y_0$ .

To every topological space  $X$  with distinguished point  $x_0$ , one can define the [fundamental group](#) based at  $x_0$ , denoted  $\pi_1(X, x_0)$ . This is the [group](#) of [homotopy](#) classes of loops based at  $x_0$ . If  $f : X \rightarrow Y$  is a [morphism](#) of pointed spaces, then every loop in  $X$  with base point  $x_0$  can be [composed](#) with  $f$  to yield a loop in  $Y$  with base point  $y_0$ . This [operation](#) is compatible with the [homotopy equivalence relation](#) and the [composition](#) of loops, and we get a [group homomorphism](#) from  $(X, x_0)$  to  $(Y, y_0)$ . We thus obtain a [functor](#) from the [category](#) of pointed topological spaces to the [category](#) of [groups](#).

In the [category](#) of topological spaces (without distinguished point), one considers [homotopy](#) classes of generic curves, but they cannot be [composed](#) unless they share an endpoint. Thus one has the fundamental groupoid instead of the [fundamental group](#), and this construction is [functorial](#).

## Chapter 90

# Algebra of continuous function

A [contravariant functor](#) from the [category](#) of topological spaces (with continuous maps as [morphisms](#)) to the [category](#) of real [associative algebras](#) is given by assigning to every topological space  $X$  the [algebra](#)  $C(X)$  of all real-valued continuous [functions](#) on that space. Every continuous map  $f : X \rightarrow Y$  induces an [algebra homomorphism](#)  $C(f) : C(Y) \rightarrow C(X)$  by the rule  $C(f)(g) = g \circ f$  for every  $g$  in  $C(Y)$ .

## Chapter 91

# Tangent and cotangent bundle

The map which sends every differentiable manifold to its tangent bundle and every smooth map to its derivative is a [covariant functor](#) from the [category](#) of differentiable manifolds to the [category](#) of vector bundles.

Doing this constructions pointwise gives the tangent space, a [covariant functor](#) from the [category](#) of pointed differentiable manifolds to the [category](#) of real vector spaces. Likewise, cotangent space is a [contravariant functor](#), essentially the [composition](#) of the tangent space with the [dual](#) space above.



## Chapter 92

# Group action / representation

Every [group](#)  $G$  can be considered as a [category](#) with a single [object](#) whose [morphisms](#) are the elements of  $G$ . A [functor](#) from  $G$  to [Set](#) is then [nothing](#) but a [group](#) action of  $G$  on a particular [set](#), i.e. a  $G$ -[set](#). Likewise, a [functor](#) from  $G$  to the [category](#) of vector spaces,  $\text{Vect}_K$ , is a [linear](#) representation of  $G$ . In general, a [functor](#)  $G \rightarrow C$  can be considered as an "action" of  $G$  on an [object](#) in the [category](#)  $C$ . If  $C$  is a [group](#), then this action is a [group homomorphism](#).

## Chapter 93

# Lie algebra

Assigning to every real (complex) Lie [group](#) its real (complex) [Lie algebra](#) defines a [functor](#).

## Chapter 94

# Tensor product

If  $\mathcal{C}$  denotes the [category](#) of vector spaces over a fixed field, with [linear](#) maps as [morphisms](#), then the [tensor product](#)  $V \otimes W$  defines a [functor](#)  $\mathcal{C} \times \mathcal{C} \rightarrow \mathcal{C}$  which is [covariant](#) in both arguments.

## Chapter 95

# Forgetful functor

The functor  $U : \mathbf{Grp} \rightarrow \mathbf{Set}$  which maps a group to its underlying set and a group homomorphism to its underlying function of sets is a functor.[8] Functors like these, which "forget" some structure, are termed forgetful functors. Another example is the functor  $\mathbf{Rng} \rightarrow \mathbf{Ab}$  which maps a ring to its underlying additive abelian group. Morphisms in  $\mathbf{Rng}$  (ring homomorphisms) become morphisms in  $\mathbf{Ab}$  (abelian group homomorphisms).

## Chapter 96

# Free functor

Going in the opposite direction of [forgetful functors](#) are free [functors](#). The [free functor](#)  $F : \mathbf{Set} \rightarrow \mathbf{Grp}$  sends every [set](#)  $X$  to the free [group](#) generated by  $X$ . [Functions](#) get mapped to [group](#) homomorphisms between free [groups](#). Free constructions exist for many [categories](#) based on structured [sets](#). See [free object](#).

## Chapter 97

# Homomorphism group

To every pair  $A, B$  of abelian groups one can assign the abelian group  $\text{Hom}(A, B)$  consisting of all group homomorphisms from  $A$  to  $B$ . This is a functor which is contravariant in the first and covariant in the second argument, i.e. it is a functor  $\text{Ab}^{\text{op}} \rightarrow \text{Ab}$  (where  $\text{Ab}$  denotes the category of abelian groups with group homomorphisms). If  $f : A_1 \rightarrow A_2$  and  $g : B_1 \rightarrow B_2$  are morphisms in  $\text{Ab}$ , then the group homomorphism  $\text{Hom}(f, g) : \text{Hom}(A_2, B_1) \rightarrow \text{Hom}(A_1, B_2)$  is given by  $g \circ f$ . See Hom functor.

## Chapter 98

# Representable functor

We can generalize the previous example to any [category](#)  $C$ . To every pair  $X, Y$  of [objects](#) in  $C$  one can assign the [set](#)  $\text{Hom}(X, Y)$  of [morphisms](#) from  $X$  to  $Y$ . This defines a [functor](#) to [Set](#) which is [contravariant](#) in the first [argument](#) and [covariant](#) in the second, i.e. it is a [functor](#)  $\text{Cop} \rightarrow C \rightarrow \text{Set}$ . If  $f : X_1 \rightarrow X_2$  and  $g : Y_1 \rightarrow Y_2$  are [morphisms](#) in  $C$ , then the [group homomorphism](#)  $\text{Hom}(f, g) : \text{Hom}(X_2, Y_1) \rightarrow \text{Hom}(X_1, Y_2)$  is given by  $g \circ f$ .

[Functors](#) like these are called representable [functors](#). An important goal in many settings is to determine whether a given [functor](#) is representable.

## Chapter 99

# Corecursion



## Chapter 100

# Coinduction

*proper definition*

\* [dual](#) to induction.

Generalises to [corecursion](#).

## Chapter 101

# Initial algebra of an endofunctor

## Chapter 102

# Terminal coalgebra for an endofunctor

# Part IV

## Citations

---

"One of the finer points of the Haskell community has been its propensity for recognizing [abstract](#) patterns in code which have well-defined, lawful representations in mathematics." (Chris Allen, Julie Moronuki - "Haskell Programming from First Principles" (2017))

Part V

Good code

## Chapter 103

# Good: Type aliasing

Use [data type](#) aliases to differentiate logic of values.

## Chapter 104

# Good: Type wideness

Wider the [type](#) the more it is [polymorphic](#), means it has broader [application](#) and fits more [types](#).

The more constrained system has more usefulness.

Unconstrained means most flexible, but also most useless.



## Chapter 105

### Good: Print

```
print :: Show a => a -> IO ()  
print a = putStrLn (show a)
```

## Chapter 106

### Good: Fold

`foldr spine recursion` intermediated by the folding. Can terminate at any point.

`foldl spine recursion` is unconditional, then folding starts. Unconditionally recurses across the whole `spine`, if it infinite - infinitely.

## Chapter 107

# Good: Computation model

Model the [domain](#) and [types](#) before thinking about how to write computations.

## Chapter 108

Good: Make bottoms only local

## Chapter 109

Good: Newtype wrap is ideally transparent for compiler and does not change performance

## Chapter 110

Good: Instances of types/type classes must go with code you write

## Chapter 111

Good: Functions can be abstracted  
as arguments

## Chapter 112

Good: Infix operators can be bind  
to arguments



## Chapter 113

# Good: Arbitrary

Product types can be tested as a product of random generators.

Sum types require to implement generators with separate constructors, and picking one of them, use oneof or frequency to pick generators.

## Chapter 114

# Good: Principle of Separation of concerns

## Chapter 115

# Good: Function composition

In Haskell inline [composition](#) requires:

```
h.g.f $ i
```

[Function application](#) has a higher [priority](#) than [composition](#). That is why parentheses over [argument](#) are needed.

This [precedence](#) allows idiomatically [compose partially applied functions](#).

But it is a way better then:

```
h (g (f i))
```

## Chapter 116

# Good: Point-free

Use `Tacit` very carefully - it hides `types` and harder to change code `where` it is used.  
Use just enough `Tacit` to communicate a bit better. Mostly only partial `point-free` communicates better.

### 116.1 Good: Point-free is great in multi-dimensions

BigData and OLAP analysis.

## Chapter 117

# Good: Functor application

[Function application](#) on n levels beneath:

```
(fmap . fmap) function twoLevelStructure
```

How `fmap . fmap` [typechecks](#):

```
(.)      :: (b -> c) -> (a -> b) -> a -> c
fmap     :: Functor f => (m -> n) -> f m -> f n
fmap     :: Functor g => (x -> y) -> g x -> g y

fmap . fmap :: (Functor f, Functor g)
            => ((g x -> g y) -> f . g x -> f . g y)
            -> (( x -> y) -> g x -> g y)
            -> ( x -> y) -> f . g x -> f . g y
fmap . fmap :: (x -> y) -> f . g x -> f . g y
```

## Chapter 118

# Good: Parameter order

In [functions parameter order](#) is important.

It is best to use first the most reusable [parameters](#).

And as last one the one that can be the most [variable](#), that is important to chain.

## Chapter 119

# Good: Applicative monoid

There can be more than one valid [Monoid](#) for a [data type](#). &&

There can be more than one valid [Applicative](#) instance for a [data type](#). ->

There can be different [Applicatives](#) with different [Monoid](#) implementations.

## Chapter 120

### Good: Creative process

- 120.1 Pick philosophy principles one to three the more - the harder the implementation
- 120.2 Draw the most blurred representation
- 120.3 Deduce **abstractions** and write remotely what they are
- 120.4 Model of computation
  - 120.4.1 Model the **domain**
  - 120.4.2 Model the **types**
  - 120.4.3 Think how to write computations
- 120.5 Create



## Chapter 121

«<Good: About operators (<\$ ) ( \*\*>) (<\* ) (» ) »>

[Where](#) character is not present - discard the according processing of a [parameter](#).  
(» ) is an [exception](#), it does the reverse. ignores the first [parameter](#).

## Chapter 122

Good: About functions like `{mapM, sequence}_`

Trailing `_` means ignoring the result.

# Chapter 123

## Good: Guideliles

### 123.1 Wiki.haskell

#### 123.1.1 Documentation

123.1.1.1 Comments write in **application** terms, not technical.

123.1.1.2 Tell what code needs to do not how it does.

#### 123.1.2 Haddock

123.1.2.1 Put haddock comments to ever exposed **data type** and **function**.

123.1.2.2 Haddock header

```
{- |  
Module      : <File name or $Header$ to be replaced automatically>  
Description : <optional short text displayed on contents page>  
Copyright   : (c) <Authors or Affiliations>  
License     : <license>  
  
Maintainer  : <email>  
Stability   : unstable | experimental | provisional | stable | frozen  
Portability : portable | non-portable (<reason>  
  
<module description starting at first column>  
-}
```

#### 123.1.3 Code

123.1.3.1 Try to stay closer to portable (Haskell98) code

123.1.3.2 Try make lines no longer 80 chars

123.1.3.3 Last char in file should be newline

123.1.3.4 Symbolic **infix** identifiers is only library writer right

123.1.3.5 Every **function** does one thing.

## Chapter 124

# Good: Use Typed holes to progress the code

[Typed holes](#) help build code in complex situations.

## Chapter 125

Good: Haskell allows infinite terms  
but not infinite types

That is why infinite [types](#) throw infinite [type error](#).

## Chapter 126

# Good: Use type synonyms to differ the information

Even if there is `types` - define `type` synonyms. They are free.

That distinction with synonyms, would allow `TypeSynonymInstances`, which would allow to create a different `type class` instances and behaviour for different information.

## Chapter 127

«<Good: Control.Monad.Error ->  
Control.Monad.Except» >

## Chapter 128

# Good: Monad OR Applicative

### 128.0.1 Start writing [monad](#) using 'return', 'ap', '[liftM](#)', '[liftM2](#)', '»' instead of 'do', '»='

If you wrote code and really needed only those - move that code to [Applicative](#).

```
return -> pure
ap -> <*>
liftM -> liftA -> <$>
>> -> *>
```

### 128.0.2 Basic [case](#) when [Applicative](#) can be used

Can be rewritten in [Applicative](#):

```
func = do
  a <- f
  b <- g
pure (a, b)
```

Can't be rewritten in [Applicative](#):

```
somethingdoSomething' n = do
  a <- f n
  b <- g a
pure (a, b)
```

(f n) creates [monadic structure](#), [binds](#) it to *a* which is consumed then by *g*.

### 128.0.3 [Applicative](#) block vs [Monad](#) block

With [Type Applicative](#) every condition fails/succeeds independently. It needs a boilerplate [data constructor](#)/value pattern matching code to work. And code you can write only for so many cases and [types](#), so boilerplate can not be so flexible as [Monad](#) that allows [polymorphism](#).

With [Type Monad](#) computation can return value that dependent from the previous computation result. So abort or dependent processing can happen.



## Chapter 129

# Good: Haskell Package Versioning Policy

Version policy and dependency management.

[width=.9]Good<sub>code</sub>/pvp - decision - tree<sub>2</sub>019 - 06 - 17<sub>1</sub>5 - 49 - 21

### 129.1 \*

PVP

Good: PVP

## Chapter 130

# Good: Linear type

Linear types are great to control/minimize resource usage.

## Chapter 131

# Good: Exception vs Error

Many languages and Haskell have it all mixup. Here is table showing what belongs to one or other in standard libraries:

---

<a href="#">Exception</a>	Prelude.catch, Control. <a href="#">Exception</a> .catch, Control. <a href="#">Exception</a> .try, IOError, Control. <a href="#">Monad</a> . <a href="#">Error</a>
<a href="#">Error</a>	<a href="#">error</a> , assert, Control. <a href="#">Exception</a> .catch, Debug.Trace.trace

## Chapter 132

### Good: Let vs. Where

`let ... in ...` is a separate [expression](#). In contrast, `where` is [bound](#) to a surrounding syntactic [construct](#) (namespace).

## Chapter 133

# Good: RankNTypes

Can powerfully synergyze with [ScopedTypeVariables](#).

## Chapter 134

# Good: Orphan type instance

Practice to address orphan instances:

Does [type class](#) or [type](#) defined by you:

<a href="#">Type class</a>	<a href="#">Type</a>	Recommendation
✓	✓	{ <a href="#">Type</a> , instance} in the same <a href="#">module</a> { <a href="#">Typeclass</a> & instance} in the same <a href="#">module</a> {Define newtype wrap, its instances} in the same <a href="#">module</a>

## Chapter 135

# Good: Smart constructor

Only proper smart [constructors](#) should be exported. Do not export [data type constructor](#), only a [type](#).

## Chapter 136

### Good: Thin category

In \* all [morphisms](#) are [epimorphisms](#) and [monomorphisms](#).



## Chapter 137

# Good: Recursion

Writing/thinking about [recursion](#):

- a.* Find the base cases, on input of which the answer can be provided right away. There is mostly one [base case](#), but sometimes there can be several of them. Typical base cases are: [zero](#), the empty [list](#), the empty tree, null, etc.
- b.* Do inductive [case](#). The [recursive](#) invocation. The [argument](#) of a [recursive](#) call needs to be smaller than the current [argument](#). So it would be gradually closer to the [base case](#). The idea is that processes eventually hits the [base case](#).

Simple functional [application](#) is used in the [recursion](#).

Assume that the [functions](#) would return the right result.

## Chapter 138

# Good: Monoid

<>:

[Sets](#) - union.

Maps - left-biased union.

Number - `Sum`, `Product` form separate [monoid categories](#).

## Chapter 139

# Good: Free monad

The main [case](#) of usage of Free [monads](#) in Haskell:

Start implementation of the [monad](#) from a Free [monad](#), drafting the base [monadic](#) operations, then add custom operations.

Gradually build on top of Free [monad](#) and try to find homomorphisms from [monad](#) to [objects](#), and if only [objects](#) are needed - get rid of the free [monad](#).

## Chapter 140

Good: Use mostly where clauses

## Chapter 141

Good: Where clause is in a scope  
with function parameters

## Chapter 142

Good: Strong preference towards pattern matching over {head, tail, etc.} functions

head and tail and alike [functions](#) are often partial ([unsafe](#)) functions.

## Chapter 143

# Good: Patternmatching is possible on monadic bind in do

Example:

```
instance (Monad m) => Functor (StateT s m) where
  fmap f m = StateT $ \s -> do
    (x, s') <- runStateT m s  -- Here is a pattern matching bind
    return (f x, s')
```

## Chapter 144

# Good: Applicative vs Monad

Giving not `Monad` but `Applicative` requirement allows parallel computation, but if there should be a chaining of the intermediate state - it must be [monadic](#).



## Chapter 145

# Good: StateT, ReaderT, WriterT

Reader trait: (r ->).

Writer trait: (a, w).

State trait is combination of both:

```
newtype StateT s m a =  
  StateT { runStateT :: s -> m (a, s) }
```

```
newtype ReaderT r m a =  
  ReaderT { runReaderT :: r -> m a }
```

```
newtype WriterT w m a =  
  WriterT { runWriterT :: m (a, w) }
```

State trait fully replaces writer.

## Chapter 146

# Good: Working with MonadTrans and lift

From the `lift . pure = pure` follows that `MonadTrans` [type](#) can have a `pure` defined with `lift`.

Stacking of `MonadTrans` [monads](#) can result in a lot of chained `lift` and `unwraps`. There is many ways to cope with that but the most robust and common is to [abstract](#) representation with `newtype` on the `Monad` [stack](#). This can reduce caining or remove the manual [lifting](#) withing the [Monad](#).

For perfect combination for contributors to be able to extend the code - keep the `Internal` [module](#) that has a raw representation.

## Chapter 147

# Good: Don't mix Where and Let

`let` and `where` create a [recursive set](#) of definitions with can explode, don't mix them together in code.

## Chapter 148

# Good: Where vs. Let

**Let** is self-recursive lazy pattern. It is checked and errors only at execution time. Binds only inside expression it is binded to.

**Where** is a part of definition, scoped over definition implemetations and guards, not self-recursive.

## Chapter 149

# Good: The proper nature algorithm that models behaviour of many objects is computation heavy

God does not care about our mathematical difficulties. He integrates empirically.

One who is found of mathematical meaning loves to [apply](#) it. But if we implement the "real" algorithms behind nature processes, we face the need to go through the computations of laws of all particles.

Computation of nature is always a middle way between ideal theory behaviour and computation simplification.

## Chapter 150

Good: In Haskell parameters bound by lambda declaration instantiate to only one concrete type

Because of [let-bound polymorphism](#):

This is illegal in Haskell:

```
foo :: (Int, Char)
foo = (\f -> (f 1, f 'a')) id
```

Lambda-bound function (i.e., one passed as [argument](#) to another [function](#)) cannot be instantiated in two different ways, if there is a [let-bound polymorphism](#).

## Chapter 151

# Good: Instance is a good structure to draw a type line

Instances for [data type](#) can differentiate by [constraints](#) & [types](#) of arguments. So instance can preserve [type](#) boundary, and [data type declaration](#) can stay very [polymorphic](#). If the need to extend the [type](#) boundaries arrive - the instances may extend, or new instances are created, while used [data type](#) still the same and unchanged.

## Chapter 152

# Good: MTL vs. Transformers

Default of `mtl`.

`Transformers` is Haskell-98, doesn't have functional dependencies, lacks the `monad` classes, has manual `lift` of operations to the composite `monad`.

MTL extends `transformers`, providing more instances, features and possibilities, may include `alternative` packages features as `mtl-tf`.



Part VI

Bad code

## Chapter 153

# Bad pragma

### 153.1 Bad: Dangerous **LANGUAGE pragma** option

- [DatatypeContexts](#)
- `OverlappingInstances`
- `IncoherentInstances`
- `ImpredicativeTypes`
- `AllowAmbiguousTypes`
- [UndecidableInstances](#) - often

## Part VII

Useful **functions** to remember

# Chapter 154

## Prelude

```
enumFromTo
enumFromThenTo
reverse
show :: Show a => a -> String
flip
sequence - Evaluate each monadic action in the structure from left to right, and collect the results
:sprint - show variables to see what has been evaluated already.
minBound - smaller bound
maxBound - larger bound
cycle :: [a] -> [a] - indefinitely cycle s list
repeat - indefinitely list from value
elemIndex e l - return first index, returns Maybe
fromMaybe (default if Nothing) e :: Maybe a -> a
lookup :: Eq a => a -> [(a, b)] -> Maybe b
```

### 154.1 Ord

```
compare
```

### 154.2 Calc

div - always makes rounding down, to infinity  
divMod - returns a [tuple](#) containing the result of integral division and modulo

### 154.3 [List](#) operations

```
concat - [ [a] ] -> [a]
elem x xs - is element a part of a list
zip :: [a] -> [b] -> [(a, b)] - zips two lists together. Zip stops when one list runs out.
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c] - do the action on corresponding elements of list a
```

## Chapter 155

### Data.List

`intersperse :: a -> [a] -> [a]` - gets the value and incerts it between values in array  
`nub` - remove duplicates from the list

## Chapter 156

# Data.Char

```
ord (Char -> Int)
chr (Int -> Char)
isUpper (Char -> Bool)
toUpper (Char -> Char)
```

## Chapter 157

# QuickCheck

```
quickCheck :: Testable prop => prop -> IO ()  
quickCheck . verbose - run verbose mode
```

## Part VIII

# Tools



## Chapter 158

# ghc-pkg

[List](#) installed packages:

```
ghc-pkg list
```

## Chapter 159

# Search over the Haskell packages code: Codesearch from Aelve

<https://codesearch.aelve.com/>

## Chapter 160

# Integration of NixOS/Nix with Haskell IDE Engine (HIE) and Emacs (Spacemacs)

160.1 1. Install the Cachix: <https://github.com/cachix/cachix>

160.2 2. Installation of HIE: <https://github.com/infinisil/all-hies/#cached-builds>

160.2.1 2.1. Provide cached builds

```
cachix use all-hies
```

160.2.2 2.2.a. Installation on NixOS distribution:

```
{ config, pkgs, ... }:
```

```
let
```

```
    all-hies = import (fetchTarball "https://github.com/infinisil/all-hies/tarball/master") {};
```

```
in {
```

```
    environment.systemPackages = with pkgs; [
```

```
        (all-hies.selection { selector = p: { inherit (p) ghc865 ghc864; }; })
```

```
    ];
```

```
}
```

Insert your GHC versions.

Switch to new configuration:

```
sudo -i nixos-rebuild switch
```

### 160.2.3 2.2.b. Installation with Nix package manager:

```
nix-env -iA selection --arg selector 'p: { inherit (p) ghc865 ghc864; }' -f 'https://github.com/i
```

Insert your GHC versions.

## 160.3 3. Emacs (Spacemacs) configuration:

```
dotspacemacs-configuration-layers
'(

  auto-completion

  (lsp :variables
    default-nix-wrapper (lambda (args)
      (append
        (append (list "nix-shell" "-I" "." "--command" )
          (list (mapconcat 'identity args " ")))
        )
      (list (nix-current-sandbox))
      )
    )

    lsp-haskell-process-wrapper-function default-nix-wrapper
  )

  (haskell :variables
    haskell-enable-hindent t
    haskell-completion-backend 'lsp
    haskell-process-type 'cabal-new-repl
  )

)

dotspacemacs-additional-packages '(
  direnv
  nix-sandbox
)

(defun dotspacemacs/user-config ()

  (add-hook 'haskell-mode-hook 'direnv-update-environment) ;; If direnv configured

)
```

Where:

auto-completion configures YASnippet.

nix-sandbox (<https://github.com/travisbhartwell/nix-emacs>) has a great helper [functions](#). Using `nix-current-sandbox` function in `default-nix-wrapper` that used to properly configure `lsp-haskell-process-wrapper-function`.

## CHAPTER 160. INTEGRATION OF NIXOS/NIX WITH HASKELL IDE ENGINE (HIE)

### 160.4. 4. OPEN THE HASKELL FILE FROM A PROJECT AND EMACS (SPACEMACS)

Configuration of the `lsp-haskell-process-wrapper-function` `default-nix-wrapper` is a key for HIE to work in `nix-shell`

Inside `nix-shell` the `haskell-process-type` `'cabal-new-repl` is required.

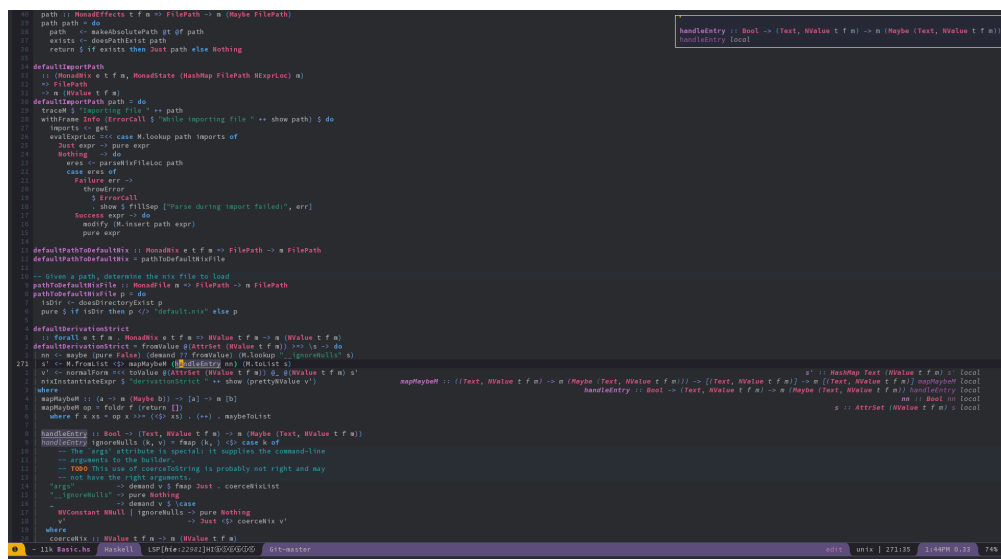
Configuration was reassembled from: <https://github.com/emacs-lsp/lsp-haskell/blob/8f2dbb6e827b1adce6360c56f795f29ecff1d7f6/lsp-haskell.el#L57> & its authors config: [\[\[https://github.com/sevanspowell/dotfiles/blob/master/spacemacs\]\]](https://github.com/sevanspowell/dotfiles/blob/master/spacemacs)

Refresh Emacs.

## 160.4 4. Open the Haskell file from a project

Open system monitor, observe the process of environment establishing, packages loading & compiling.

## 160.5 5. Be pleased writing code



Now, the powers of the Haskell, Nix & Emacs combined. It's fully in your hands now. Be cautious - you can change the world.

## 160.6 6. (optional) Debugging

a. If receiving sort-of:

`readCreateProcess : cabal-helper-wrapper failure`

HIE tries to run `cabal` operations like on the non-Nix system. So it is a problem with detection of `nix-shell` environment, running inside it.

- a.* If HIE keeps getting ready, failing & restarting - check that the projects `ghc --version` is declared in your `all-hie` NixOS configuration.

# Chapter 161

## Debugger

Provides:

- [set](#) a breakpoints
- observe step-by-step [evaluation](#)
- tracing mode

Breakpoints

```
:break 2
:show breaks
:delete 0
:continue
```

Step-by-step

```
:step main
```

[List](#) information at the breakpoint

```
:list
```

What been evaluated already

```
:sprint name
```

## Chapter 162

# GHCID

Commands to run the compile/check loop:

`cabal > 3.0 command:`

```
ghcid --command='cabal v2-repl --repl-options=-fno-code --repl-options=-fno-break-on-exception --g
```

`cabal < 3.0 command:`

```
ghcid --command='cabal new-repl --ghc-options=-fno-code --ghc-options=-fno-break-on-exception --g
```

`nix-shell cabal > 3.0 command:`

```
nix-shell --command 'ghcid --command="cabal v2-repl --repl-options=-fno-code --repl-options=-fno-
```

`nix-shell cabal < 3.0 command:`

```
nix-shell --command 'ghcid --command="cabal new-repl --ghc-options=-fno-code --ghc-options=-fno-b
```



## Part IX

# Libs

## Chapter 163

# Exceptions

- 163.1 **Exceptions** - optionally **pure** extensible **exceptions** that are compatible with the mtl
- 163.2 **Safe-exceptions** - safe, simple API **equivalent** to the underlying implementation in terms of power, encourages best practices minimizing the chances of getting the **exception** handling wrong.
- 163.3 **Enclosed-exceptions** - capture **exceptions** from the enclosed computation, while reacting to asynchronous **exceptions** aimed at the calling thread.

## Chapter 164

# Memory management

### 164.1 membrain - [type](#)-safe memory units

## Chapter 165

# Parsers - megaparsec

## Chapter 166

### CLIs - `optparse-`[applicative](#)

## Chapter 167

### HTML - Lucid

## Chapter 168

# Web applications - Servant

## Chapter 169

### IO libraries

- 169.1 Conduit - practical, monolythic, guarantees termination return
- 169.2 Pipes + Pipes Parse - modular, more primitive, theoretically driven



## Chapter 170

### JSON - aeson

## Chapter 171

# Backpack

On 1-st compilation - `*`  analyzes the `abstract` signatures without loading side modules, doing the `type check` with assumption that modules provide right `type` signatures, the process does not emit any `binary` code and stores the intermediate code in a special form that allows flexibly connect modules provided. Which allows later to compile project with particular instantiations of the modules. Major work of this process being done by internal Cabal `*`  support and `*`  system that modifies the intermediate code to fit the `module`.

Part X

Drafts

## Chapter 172

# Exception handling

**Exception** must include all **context** information that may be useful.

Store information in a form for further probable deeper automatic diagnostic.

Sensitive data/dummies for it - can be useful during development.

Sensitive data should be stripped from a program logging & **exceptions**.

**Exception** system should be extendable, data storage & representation should be easily extendable.

**Exception** system should allow easy exhaustive checking of **errors**, since the different **errors** can happen.

**Exception** system should be automatically well-documented and transparent.

**Exception** system should have controllable breaking changes downstream.

**Exception** system should allow complex composite (**sets**) **exceptions**.

**Exception** system should be lightweight on the **type** signatures of other **functions**.

**Exception** system should automate the collection of **context** for a **exception**.

**Exception** system should have **properties** and according **functions** for particular **types** of **errors**.

**String** is simple and convenient to throw **exception**, but really a mistake because it the most cumbersome choice:

- Any **Exception** instance can be converted to a **String** with **either** **show** or **displayException**.
- Does not include key debugging information in the **error** message.
- Does not allow developer to access/manage the **Exception** information.
- **Exception** messages need to be constructed ahead of time, it can not be internationalized, converted to some data/file format.
- **Exception** can have a sensitive information that can be useful for developer during work, but should not be logged/shown to end-user. Stripping it from **Strings** in the changing project is a hard task.
- Impossible to rely on this representation for further/deeper inspection.
- Impossible to have exhaustive checking - no knowledge no check, no warning if some cases are not handled.

Universal [exception type](#):

- Able to inspect every possible [error case](#) with [pattern match](#).
- Self-documenting. Shows the hierarchical system of all [exceptions](#).
- Transparent. Ability to discern in current situation what [exceptions](#) can happen
- New [exception constructor](#) causes breaking change to downstream.
- Wrongly implies completeness. Untreated [Errors](#) can happen, different [exception](#) can arrive from the outside code.

Sum [type](#) must be separate, and [product type structure](#) over it.

Separate [exception type](#) of

Individual [exception types](#):

- Writing & seing & working with exactly what will go wrong because there is only one possible [error](#) for this [type](#) of [exception](#). [Pattern match](#) happens only onconditions, [constructors](#) that should happen.
- Knowledge what exectly goes wrong allows wide usage of [Either](#).
- It is hard to handle complex [exceptions](#) in the unitary system. Real wrorld can return not a particular [case](#), but a [set](#) of cases {[object](#) not found, path is unreachable, access is denied}.
- [Type](#) signatures grow, and even can become complex, since every [case](#) of [exception](#) has its own [type](#).
- Impure [throw](#) that users can/should use for your code must account for all your [exception types](#).

[Abstract exception type](#):

[Exception type](#) entirely opaque and inspectable only by accessor [functions](#).

- Updating the internals without breaking the API
- Semi-automates the [context](#) of [exception](#) with passing it to accessors.
- Predicates can be [applied](#) to more than one [constructor](#). Which are [properties](#) that allows to make complex [exceptions](#) much easier to handle.
- Not self-documenting.

- Possible options by design are hidden from the downstream, documentation must be kept.
- When you change the [exception](#) handling/throwing [errors](#) it does not show to the downstream.

Composit approach:

Provide the [set](#) of [constructors](#) and also a [set](#) of predicates and [set](#) of accessors.

Use [pattern synonyms](#) to provide a documented accessor [set](#) without exposing internal [data type](#).

In GHC 8.8 the change was made:

The fail method of [Monad](#) has been removed in favor of the method of the same name in the MonadFail class.

MonadFail(..) is now exported from the Prelude and Control.[Monad](#) modules.

The MonadFailDesugaring language extension is now deprecated, as its effects are always enabled.

So:

```
import           Control.Monad.Fail
...
class MonadFail m => MonadFile m
...
-- use error instead of fail
Nothing         -> error ("Message " <> show x)
```

# Chapter 173

## Constraints

Very strong Haskell [type](#) system makes possible to work with code from the top down, an [axiomatic semantics](#) approach, from [constraints](#) into [types](#).

- Helps to form the [type level](#) code (aka [join](#) points of the code).
- Uses the piling up of [constraints/types](#) information. At some point pick and satisfy [constraints](#), can be done one at a time.
- Provides hints through [type level](#) formulation for [term level](#) calculations, does not formulate the [term level](#).
- Tedious method (a lot of boilerplate and rewriting it) but pretty simple and relaxing.
- [Set](#) of [constraints](#).
- When it is needed or convenient, single [constraint](#) gets a little more realistically concrete/abstracted.

Main [type](#) detail annotation thread can happen in [main](#) or special wrapper [function](#), localization is inside [functions](#).

a. Rest of [constraints set](#) shifts to source [type](#).

3.a. For the class handled or known how to handle - write a [base case](#) instance description.

```
instance (Monad m) => MonadReader r (ReaderT r m)
```

3.b. For others write [recursive](#) instance descriptions:

All other unsolved [constraints](#) move into the source [polymorphic variable](#).

```
instance (MonadError e m) => MonadError e (ReaderT r m)
instance (MonadState s m) => MonadState s (ReaderT r m)
```

- a.* Repeat from 1 until considered done.
- b.* Code condensed into terse form.

`MonadError constraints` is `IOException`, not for the `String`. `IOException` vs `String`.

Reverse pluck `MonadReader constraint` with `runReader` on the `object`.

`MonadState - StateT`



## Chapter 174

# Monad transformers and their type classes

## Chapter 175

# Layering **monad** transformers

Different layering of the same **monad** transformers is functionality is the same, but the form is different. Surrounding handling **functions** would need to be different.

# Chapter 176

## Hoogle

### 176.1 Search

Text search (case insensitive):

- `a`
- `map`
- `con map`

Type search:

- `:: a`
- `:: a -> a`

Text & type:

`=id a -> a=`

### 176.2 Scope

#### 176.2.1 Default

Scope is Haskell Platform (and Haskell keywords).

All Hackage packages are available to search with:

#### 176.2.2 Hierarchical module name system (from big letter):

- `fold +Data.Map` finds results in the `Data.Map` module

- `file -System` excludes results from modules such as `System.IO`, `System.FilePath.Windows` and `Distribution.System`

### 176.2.3 Packages (lower **case**):

- `mode +platform`
- `mode +cmdargs` (only)
- `mode +platform +cmdargs`
- `file -base` (Haskell Platform, excluding the "base" package)

## Chapter 177

# ST-Trick monad

ST is like a [lexical scope](#), where all the [variables](#)/state disappear when the [function](#) returns  
[https://wiki.haskell.org/ST\\_monad](https://wiki.haskell.org/ST_monad)  
<https://www.schoolofhaskell.com/school/to-infinity-and-beyond/older-but-still-interesting/deamortized-strg/Monad/ST>  
<https://dev.to/jvanbruegge/what-the-heck-is-polymorphism-nmh>

### 177.1 \*

ST-Trick

## Chapter 178

# Either

Allows to separate and preserve information about happened, ex. [error](#) handling.

### 178.1 \*

Either data type

## Chapter 179

# Inverse

- a.* [Inverse function](#)
- b.* In logic:  $P \rightarrow Q \Rightarrow \neg P \rightarrow \neg Q$ , & same for [category duality](#).
- c.* For [operation](#): element that allows reversing [operation](#), having an element that with the [dual](#) produces the [identity](#) element.
- d.* See [Inversion](#).

## Chapter 180

# Inversion

- a.* Is a [permutation where](#) two elements are out of [order](#).
- b.* See [Inverse](#)



## Chapter 181

# Inverse function

$$f_{x \rightarrow y} \circ (f_{x \rightarrow y})^{-1} = 1_x$$

\*  $\iff$  function is bijective.

Otherwise - partial inverse

## Chapter 182

# Inverse morphism

For  $f : x \rightarrow y$ :

$\exists g : g \circ f = 1^x$  -  $g$  is left [inverse](#) of  $f$ .

$\exists g : f \circ g = 1^y$  -  $g$  is right [inverse](#) of  $f$ .

## Chapter 183

# Partial inverse

\* when [function](#) is now [bijective](#). When [bijective](#) see [inverse function](#).

## Chapter 184

# PatternSynonyms

Enables [pattern synonym declaration](#), which always begins with the `pattern` word.  
Allows to [abstract](#)-away the [structures](#) of pattern matching.

### 184.1   \*

Pattern synonym  
Pattern synonyms

## Chapter 185

# GHC debug keys

### 185.1 -ddump-ds

Dump desugarer output.

#### 185.1.1 \*

Desugar  
GHC desugar

## Chapter 186

# GHC optimize keys

### 186.1 -foptimal-applicative-do

$O(n^3)$

Always finds optimal [reduction](#) into  $\langle * \rangle$  for [ApplicativeDo](#) do notation.

## Chapter 187

# Computational trinitarianism

Taken from: <https://ncatlab.org/nlab/show/computational+trinitarianism>

Under the [statements](#):

- [propositions](#) as [types](#)
- programs as proofs
- [relation](#) between [type](#) theory and [category](#) theory

the following notions are [equivalent](#):

== [proposition](#) proof (Logic)

== generalized element of an [object](#) ([Category](#) theory)

== typed program with output ([Type](#) theory & Computer science)

Table 187.1: [Computational trinitarianism](#)

Logic	Category theory	Type theory
true	terminal object / (-2)-truncated object	h-level 0-type
false	initial object	empty type
proposition	(-1)-truncated object	h-proposition
proof	generalized element	program
cut rule	composition of classifying morphisms / pullback of display maps	substitution
cut elimination for implication	counit for hom-tensor adjunction	beta reduction
introduction rule for implication	unit for hom-tensor adjunction	eta conversion
logical conjunction	product	product type
disjunction	coproduct ((-1)-truncation of)	sum type (br)
implication	internal hom	function type
negation	internal hom into initial object	function type
universal quantification	dependent product	dependent product
existential quantification	dependent sum ((-1)-truncation of)	dependent sum
equivalence	path space object	identity type
equivalence class	quotient	quotient type

Continued from previous page		
Logic	Category theory	Type theory
induction	colimit	inductive type
higher induction	higher colimit	higher induction
completely presented set	discrete object/0-truncated object	h-level 2-type
set	internal 0-groupoid	Bishop set/set
universe	object classifier	type of types
modality	closure operator, (idempotent) monad	modal type theory
linear logic	(symmetric, closed) monoidal category	linear type theory
proof net	string diagram	quantum circuit
(absence of) contraction rule	(absence of) diagonal	no-cloning theorem
	synthetic mathematics	domain specification

187.1 \*

Trinitarism



## Chapter 188

# Techniques functional programming deals with the state

### 188.1 Minimizing

Do not rely on state, try not to change the state. Use it only when it is very necessary.

### 188.2 Concentrating

Concentrate the state in one place.

### 188.3 Deferring

Defer state to the last step of the program, or to external system.

## Chapter 189

# Monadic Error handling

```
(>>=) :: m a -> (a -> m b) -> m b -- A.E  A - computes and drops if error value happens.  
catch :: c a -> (e -> c a) -> c a -- E.E  A - handles "errors" as "normal" values and stops when
```

## Chapter 190

# Functions

Total [function](#) uses [domain](#) fully, but takes only part of the [codomain](#).

[Function](#) allows to collapse [domain](#) values into [codomain](#) value. Meaning the [function](#) allows to loose the information.

So total [function](#) is a computation that loses the information or into bigger codomains.

That is why the [function](#) has a directionality, and [inverse](#) total process is partially possible.

Directionality and invertability are terms.

# Chapter 191

## Void

Emptiness.

Can not be grasped, touched.

A logically uninhabited [data type](#).

(Since [basis](#) of logic is tautologically True and [Void](#) value can not be addressed - there is a logical paradox with the [Void](#)).

Is an [object](#) included into the [Hask category](#), since:

```
:t (id :: Void -> Void)
(id :: Void -> Void) :: Void -> Void

id for it exists.
```

[Type](#) system corresponds to [constructive logic](#) and not to the classical logic.

Classical logic answers the question "Is this actually true".

Constructive (Intuitionistic) logic answers the question "Is this provable".

Also has [functions](#):

```
-- Represents logical principle of explosion: from falsehood, anything follows.
absurd :: Void -> a
```

```
-- If Functor holds only Void - it holds no values.
vacuous :: Functor f => f Void -> f a
```

```
-- If Monad holds only Void - it holds no values.
vacuousM :: Monad m => m Void -> m a
```

Design pattern: use [polymorphic data types](#) and [Void](#) to get rid of possibilities when you need to.

### 191.1 \*

Nothing, Haskell [expressions](#) can't return [Void](#).

Also see: [Maybe](#).

## Chapter 192

# Constructive proof

Method of proof that demonstrates the existence of a mathematical [object](#) by creating or providing a method for creating the [object](#).

## Chapter 193

# Intuitionistic logic

[Proposition](#) considered **True** due to direct evidence of existence through constructive proof using [Curry-Howard isomorphism](#).

\* does not include classic logic fundamental axioms of the excluded middle and double negation elimination. Hence \* is weaker then classical logic. Classical logic includes \*, all theorems of \* are also in classical logic.

### 193.1 \*

Constructive logic

## Chapter 194

# Principle of explosion

From asserted [statement](#) that contains contradiction - anything can be proven.  
Ancient principle of logic. Both in classical & intuitionistic logic.

### 194.1 \*

Ex falso quodlibet

Ex falso sequitur quodlibet

EFG

Ex contradictione quodlibet

Ex contradictione sequitur quodlibet

ECQ

Deductive explosion

Pseudo-Scotus

## Chapter 195

# Universal **property**

A **property** of some construction which boils down to (is manifestly **equivalent** to) the **property** that an associated **object** is a universal **initial object** of some (auxiliary) **category**.



## Chapter 196

# Yoneda lemma

Allows the embedding of any [category](#) into a [category](#) of [functors](#) ([contravariant set-valued functors](#)) defined on that [category](#). It also clarifies how the embedded [category](#), of representable [functors](#) and their [natural transformations](#), relates to the other [objects](#) in the larger [functor category](#).

The Yoneda lemma suggests that instead of studying the (locally small) [category](#)  $C \{\{\{C\}\}\}\mathcal{C}$ , *one should study the [category](#)*

## Chapter 197

# Monoidal category, functoriality of ADTs, Profunctors

Category equipped with tensor product.

$\langle \rangle$

wich is a functor for  $*$ .

Set category can be monoidal under both product (having terminal object) or coproduct (having initial object) operations, if according operation exist for all objects.

Any one-object category is  $*$ .

$(a, ()) \sim a$  up to unique isomorphism, which is called Lax monoidal functor.

Product and coproduct are functorial, so, since:  
Algebraic data type construction can use:

- Type constructor
- Data constructor
- Const functor
- Identity functor
- Product
- Coproduct

Any algebraic data type is functorial.

## Chapter 198

# Const functor

Maps all [objects](#) of source [category](#) into one (fixed) [object](#) of target [category](#), and all [morphisms](#) to [identity morphism](#) of that fixed [object](#).

```
instance Functor (Const c)
  where
    fmap :: (a -> b) -> Const c a -> Const c b
    fmap _ (Const c) = Const c
```

In [Category](#) theory denoted:

Last [type parameter](#) that bears the target [type](#) of lifted [function](#) ([b](#)) and is a [proxy type](#).

Analogy: the container that allways has an [object](#) attached to it, and everything that is put inside - changes the container [type](#) accordingly, and dissapears.

## Chapter 199

# Arrow in Haskell

```
(->) a b = a -> b
```

[Functorial](#) in the last [argument](#) & called Reader [functor](#).

```
newtype Reader c a = Reader (c -> a)
```

```
  fmap = ( . )
```

## Chapter 200

# Contravariant functor

```
fmap :: (a -> b) -> Op c a -> Op c b
      (a -> c) -> (b -> c)
```

$$\begin{array}{ccc} a & \longrightarrow & b \\ & \searrow & \vdots \\ & & c \end{array}$$
$$(a \rightarrow b)^C = (a \leftarrow b)^{C^{op}}$$

```
class Contravariant f
  where
    contramap :: (b -> a) -> (f a -> f b)
```

$$\begin{array}{ccc} a & \longrightarrow & b \\ & \searrow & \vdots \text{contravariant} \\ & & c \end{array}$$

If [arrows](#) does not commute Contravariant functor anyway allows to [construct](#) transformation between these such [arrows](#) to other [arrow](#).

## Chapter 201

# Profunctor

$(\multimap) \ a \ b$

$C^{op} \times C \rightarrow C$

It is called profunctor.

`dimap :: (a' -> a) -> (b -> b') -> p a b -> p a' b'`

So, profunctor in [case](#) of [arrow](#):

$$\begin{array}{ccc} a & \xleftarrow{f} & a' \\ \downarrow h & & \downarrow \text{profunctor} \\ b & \xrightarrow{g} & b' \end{array}$$

```
dimap :: (a' -> a) -> (b -> b') -> p a b -> p a' b'
dimap ::      f          g      -> (a -> b) -> (a' -> b')
dimap ::      f          g      ->      h      -> (a' -> b')
dimap = g . h . f
```

It is [contravariant functor](#) in the first [argument](#), and [covariant functor](#) in the second [argument](#).

```
dimap id <==> fmap
(flip dimap) id <==> contramap
```

## Part XI

# Reference

## Chapter 202

# Functor-Applicative-Monad Proposal

Well known event in Haskell history: [https://github.com/quchen/articles/blob/master/applicative\\_monad.md](https://github.com/quchen/articles/blob/master/applicative_monad.md).

Math justice was restored with a RETroactive CONTinuity. Invented in computer science term [Applicative](#) (lax monoidal functor) become a [superclass](#) of [Monad](#).

& that is why:

- `return = pure`
- `ap = <*>`
- `>= = *>`
- `liftM = liftA = fmap`
- `liftM* = liftA*`

Also, a side-kick - [Alternative](#) became a [superclass](#) of [MonadPlus](#). Hence:

- `mzero = empty`
- `mplus = (<|>)`

### 202.1 \*

Applicative-Monad proposal  
AMP



# Chapter 203

## Haskell-98

### 203.1 Old instance termination rules

- a.  $\forall$  class **constraint** ( $C\ t_1 \dots t_n$ ):
  - 1.1. **type variables** have occurrences  $\leq$  head
  - 1.2. **constructors+variables+repetitions**  $<$  head
  - 1.3.  $\neg$  **type functions** (**type** func **application** can expand to **arbitrary** size)
- b.  $\forall$  **functional dependencies**,  $tv_{\text{left}} \rightarrow tv_{\text{right}}$ , of the class, every **type variable** in  $S(tv_{\text{right}})$  must appear in  $S(tv_{\text{left}})$ , **where**  $S$  is the substitution mapping each **type variable** in the class **declaration** to the corresponding **type** in the instance head.

## Chapter 204

# Performance results and comparisons of **types** & solutions

Haskell performance

## Part XII

# Liturgy

---

<- *Laos* the people  
<- *ergon* work  
*leitourgia* giving back to the community

The life is beautiful.  
For all humans that make the life have more uniqueness.

This study would not be possible without mathematicians, Haskellers, scientists, creators, contributors. These people are the most fascinating in my life.

Special accolades for the guys at Serokell. They were the force that got me inspired & gave resources to seriously learn Haskell and create this pocket guide.