

# Fundamental Haskell notes

Encyclopedcal handbook for learning and undersatanding fundamentals

Anton Latukha

March 9, 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>Category theory</b>	<b>27</b>
2.1	*	27
2.2	Abelian category	28
2.2.1	*	28
2.3	Composition	29
2.3.1	*	29
2.4	Endofunctor category	29
2.5	Functor	29
2.5.1	*	30
2.5.2	Power set functor	30
2.5.2.1	*	30
2.5.2.2	Power set functor laws	30
2.5.2.3	Lift	31
2.5.2.4	Power set functor is a free monad	31
2.5.3	Functorial	31
2.5.4	Forgetful functor	31
2.5.4.1	*	31
2.5.5	Identity functor	31
2.5.6	Endofunctor	32
2.5.6.1	*	32
2.5.7	Applicative functor	32
2.5.7.1	*	32
2.5.7.2	Applicative law	33
2.5.7.3	*	33
2.5.7.4	Applicative function	33
2.5.7.5	Special applicatives	34
2.5.7.6	Monad	35
2.5.7.7	Alternative type class	48

2.5.7.8	$\ll \leq^* \geq \gg$	48
2.5.8	Monoidal functor	49
2.5.9	Fusion	49
2.5.10	$\ll \leq^{\$} \geq \gg$	49
2.5.11	Multifunctor	49
2.5.11.1	*	49
2.5.12	*	49
2.6	Hask category	50
2.6.1	*	50
2.7	Magma	50
2.7.1	Mag category	50
2.7.1.1	*	50
2.7.2	Semigroup	50
2.7.2.1	*	51
2.7.2.2	Monoid	51
2.8	Morphism	52
2.8.1	*	53
2.8.2	Homomorphism	53
2.8.2.1	*	53
2.8.3	Identity morphism	53
2.8.3.1	Identity	53
2.8.3.2	Identity function	54
2.8.4	Monomorphism	54
2.8.4.1	*	54
2.8.5	Epimorphism	54
2.8.5.1	*	54
2.8.6	Isomorphism	55
2.8.6.1	*	55
2.8.6.2	Lax	55
2.8.7	Endomorphism	55
2.8.7.1	Automorphism	55
2.8.7.2	*	56
2.8.8	Catamorphism	56
2.8.8.1	*	56
2.8.8.2	Catamorphism law	56
2.8.8.3	Anamorphism	57
2.8.9	Kernel	57
2.8.9.1	Kernel homomorphism	57
2.9	Set category	57
2.10	Natural transformation	57
2.10.1	*	59
2.10.2	Natural transformation component	59
2.10.2.1	*	59
2.10.3	Natural transformation in Haskell	59
2.10.4	Cat category	59
2.10.4.1	*	60
2.10.4.2	Bicategory	60
2.11	Category dual	60
2.11.1	Coalgebra	61
2.12	Thin category	61
2.12.1	*	61
2.13	Commuting diagram	61
2.13.1	*	61
2.14	Universal construction	61
2.14.1	*	62

2.15	Product	62
2.15.1	*	62
2.16	Coproduct	62
2.16.1	*	62
2.17	Free object	63
2.18	Internal category	63
2.19	Hom set	63
2.19.1	*	63
2.19.2	Hom-functor	63
2.19.3	Exponential object	64
2.19.3.1	*	64
2.19.3.2	Enriched category	64
<b>3</b>	<b>Data type</b>	<b>65</b>
3.1	*	65
3.2	Actual type	65
3.3	Algebraic data type	65
3.3.1	*	65
3.4	Cardinality	65
3.4.1	*	65
3.5	Data constant	66
3.6	Data constructor	66
3.7	data declaration	66
3.8	Dependent type	66
3.8.1	*	66
3.9	Gen type	66
3.10	Higher-kinded data type	66
3.10.1	*	66
3.11	newtype declaration	67
3.12	Principal type	67
3.13	Product data type	67
3.13.1	*	67
3.13.2	Sequence	67
3.13.2.1	*	68
3.13.2.2	List	68
3.14	Proxy type	68
3.15	Static typing	69
3.16	Structural type	69
3.16.1	*	69
3.17	Structural type system	69
3.17.1	*	69
3.18	Sum data type	69
3.19	Type alias	69
3.20	Type class	69
3.20.1	*	69
3.20.2	Arbitrary type class	70
3.20.2.1	Arbitrary function	70
3.20.3	CoArbitrary type class	70
3.20.3.1	*	70
3.20.4	Typeable type class	70
3.20.4.1	*	70
3.20.5	Type class inheritance	70
3.20.6	Derived instance	71
3.20.6.1	*	71
3.21	Type constant	71

3.22	Type constructor	72
3.23	type declaration	72
3.24	Typed hole	72
3.24.1	*	72
3.25	Type inference	72
3.25.1	*	72
3.26	Type class instance	72
3.27	Type rank	72
3.27.1	*	73
3.28	Type variable	73
3.29	Unlifted type	73
3.29.1	*	73
3.30	Linear type	74
3.30.1	*	74
3.31	NonEmpty list data type	74
3.32	Session type	74
3.33	Binary tree	74
3.34	Bottom value	74
3.34.1	*	75
3.35	Bound	75
3.35.1	*	75
3.36	Constructor	75
3.36.1	*	75
3.37	Context	75
3.37.1	*	75
3.38	Inhabit	75
3.39	Maybe	76
3.39.0.1	*	76
3.40	Expected type	76
3.41	ADT	76
3.42	Concrete type	76
3.43	Type punning	76
3.44	Kind	76
3.44.1	*	77
3.45	IO	77
<b>4</b>	<b>Expression</b>	<b>78</b>
4.1	*	78
4.2	Closed-form expression	78
4.3	RHS	78
4.4	LHS	78
4.5	Redex	79
4.6	Concatenate	79
4.7	Alpha equivalence	79
4.8	Ground expression	79
4.8.1	*	79
4.9	Variable	79
4.9.1	*	79
4.10	Phrase	80

<b>5</b>	<b>Function</b>	<b>81</b>
5.1	*	81
5.2	Arity	82
5.3	Bijection	82
5.3.1	*	82
5.4	Combinator	82
5.4.1	$\Psi$ -combinator	83
5.4.1.1	*	83
5.5	Function application	83
5.5.1	*	83
5.6	Function body	83
5.7	Function composition	83
5.7.1	*	84
5.8	Function head	84
5.9	Function range	84
5.10	Higher-order function	84
5.10.1	*	84
5.10.2	Fold	84
5.11	Injection	85
5.11.1	*	85
5.12	Partial function	85
5.13	Purity	85
5.13.1	*	85
5.14	Pure function	85
5.15	Sectioning	86
5.16	Surjection	86
5.16.1	*	86
5.17	Unsafe function	86
5.17.1	*	86
5.18	Variadic	86
5.19	Domain	86
5.20	Codomain	86
5.21	Open formula	87
5.22	Recursion	87
5.22.1	*	87
5.22.2	Base case	87
5.22.3	Tail recursion	87
5.22.4	Polymorphic recursion	87
5.22.4.1	*	87
5.23	Free variable	87
5.24	Closure	87
5.24.1	*	88
5.25	Parameter	88
5.25.1	*	88
5.26	Partial application	88
5.26.1	*	88
5.27	Well-formed formula	88
5.27.1	*	88
<b>6</b>	<b>Homotopy</b>	<b>89</b>
6.1	*	89

<b>7</b>	<b>Lambda calculus</b>	<b>90</b>
7.1	*	90
7.2	Lambda cube	90
7.2.1	*	91
7.3	Lambda function	91
7.3.1	*	91
7.3.2	Anonymous lambda function	91
7.3.2.1	*	91
7.3.3	Uncurry	92
7.4	$\beta$ -reduction	92
7.4.1	*	92
7.4.2	$\beta$ -normal form	92
7.4.2.1	*	92
7.5	Calculus of constructions	92
7.5.1	*	92
7.6	Curry–Howard correspondence	92
7.6.1	*	93
7.7	Currying	93
7.7.1	*	93
7.8	Hindley–Milner type system	93
7.8.1	*	93
7.9	Reduction	93
7.9.1	*	93
7.10	$\beta$ - $\eta$ normal form	94
7.10.1	*	94
7.11	$\eta$ -abstraction	94
7.11.1	*	94
7.12	Lambda expression	94
<b>8</b>	<b>Operation</b>	<b>95</b>
8.1	Constant	95
8.2	Binary operation	95
8.2.1	*	95
8.3	Operator	95
8.3.1	Shift operator	95
8.3.1.1	*	95
8.3.2	Differential operator	96
8.3.2.1	*	96
8.4	Infix	96
8.5	Fixity	96
8.5.1	*	96
8.6	Zero	97
8.7	Bind	97
8.7.1	*	97
8.8	Declaration	97
8.9	Dispatch	97
8.10	Evaluation	97
<b>9</b>	<b>Permutation</b>	<b>98</b>

<b>10 Point-free</b>	<b>99</b>
10.1 *	99
10.2 Blackbird	99
10.2.1 *	99
10.3 Swing	100
10.4 Squish	100
<b>11 Polymorphism</b>	<b>101</b>
11.1 *	101
11.2 Levy polymorphism	101
11.3 Parametric polymorphism	101
11.3.1 Rank-1 polymorphism	101
11.3.1.1 *	101
11.3.2 Let-bound polymorphism	102
11.3.3 Constrained polymorphism	102
11.3.3.1 Ad hoc polymorphism	102
11.3.4 Impredicative polymorphism	102
11.3.4.1 *	102
11.3.5 Higher-rank polymorphism	102
11.3.5.1 *	103
11.4 Subtype polymorphism	103
11.5 Row polymorphism	103
11.6 Kind polymorphism	103
11.7 Linearity polymorphism	104
<b>12 Pragma</b>	<b>105</b>
12.1 LANGUAGE pragma	105
12.1.1 LANGUAGE option	105
12.1.1.1 *	105
12.1.1.2 Useful by default	105
12.1.1.3 AllowAmbiguousTypes	105
12.1.1.4 ApplicativeDo	105
12.1.1.5 ConstrainedClassMethods	106
12.1.1.6 CPP	106
12.1.1.7 DeriveFunctor	106
12.1.1.8 ExplicitForAll	106
12.1.1.9 FlexibleContexts	106
12.1.1.10 FlexibleInstances	107
12.1.1.11 GeneralizedNewtypeDeriving	107
12.1.1.12 ImplicitParams	107
12.1.1.13 LambdaCase	107
12.1.1.14 MultiParamTypeClasses	107
12.1.1.15 MultiWayIf	108
12.1.1.16 OverloadedStrings	108
12.1.1.17 PartialTypeSignatures	108
12.1.1.18 RankNTypes	108
12.1.1.19 ScopedTypeVariables	109
12.1.1.20 TupleSections	109
12.1.1.21 TypeApplications	109
12.1.1.22 TypeSynonymInstances	109
12.1.1.23 UndecidableInstances	110
12.1.1.24 ViewPatterns	110
12.1.1.25 DatatypeContexts	110
12.1.1.26 StandaloneKindSignatures	110
12.1.1.27 PartialTypeSignatures	111



12.1.2	How to make a GHC LANGUAGE extension . . . . .	112
<b>13</b>	<b>Compositionality</b>	<b>113</b>
13.1	* . . . . .	113
<b>14</b>	<b>Referential transparency</b>	<b>114</b>
14.1	* . . . . .	114
<b>15</b>	<b>Semantics</b>	<b>115</b>
15.1	Operational semantics . . . . .	115
15.1.1	Argument . . . . .	115
15.1.1.1	Argument of a function . . . . .	115
15.1.1.2	First-class . . . . .	115
15.1.2	Relation . . . . .	116
15.1.2.1	* . . . . .	116
15.2	Denotational semantics . . . . .	116
15.2.1	Abstraction . . . . .	116
15.2.1.1	* . . . . .	116
15.2.1.2	Leaky abstraction . . . . .	116
15.2.1.3	Object . . . . .	117
15.2.2	Ambigram . . . . .	119
15.2.3	Binary . . . . .	119
15.2.4	Arbitrary . . . . .	119
15.2.5	Refutable . . . . .	119
15.2.6	Irrefutable . . . . .	119
15.2.7	Superclass . . . . .	119
15.2.8	Unit . . . . .	119
15.2.9	Nullary . . . . .	120
15.2.10	Syntax tree . . . . .	120
15.2.10.1	Abstract syntax tree . . . . .	120
15.2.10.2	Concrete syntax tree . . . . .	120
15.2.11	Stream . . . . .	120
15.2.12	Linear . . . . .	121
15.2.12.1	* . . . . .	121
15.2.13	Predicative . . . . .	121
15.2.14	Quantifier . . . . .	121
15.2.14.1	* . . . . .	121
15.2.14.2	Forall quantifier . . . . .	121
15.3	Axiomatic semantics . . . . .	122
15.3.1	Property . . . . .	122
15.3.1.1	* . . . . .	122
15.3.1.2	Associativity . . . . .	122
15.3.1.3	Left associative . . . . .	123
15.3.1.4	Right associative . . . . .	123
15.3.1.5	Non-associative . . . . .	123
15.3.1.6	Basis . . . . .	123
15.3.1.7	Commutativity . . . . .	124
15.3.1.8	Idempotence . . . . .	124
15.3.1.9	Distributive property . . . . .	124
15.3.2	Effect . . . . .	125
15.3.3	Bisimulation . . . . .	125
15.3.3.1	* . . . . .	125
15.4	Content word . . . . .	125
15.5	Ancient Greek and Latin prefixes . . . . .	125
15.5.1	* . . . . .	125

15.6	Idiom	125
15.6.1	*	125
15.7	Impredicative	127
15.8	Context-free grammar	127
15.8.1	*	127
<b>16</b>	<b>Set</b>	<b>128</b>
16.1	*	128
16.2	Closed set	128
16.3	Power set	128
16.4	Singleton	128
16.5	Russell's paradox	128
16.6	Cartesian product	129
16.6.1	Pullback	129
16.6.1.1	*	129
<b>17</b>	<b>Testing</b>	<b>130</b>
17.1	Property testing	130
17.1.1	Function property	130
17.1.2	Property testing type	130
17.1.3	Generator	130
17.1.3.1	*	131
17.1.3.2	Custom generator	131
17.1.4	Reusing test code	131
17.1.4.1	Test Commutative property	131
17.1.4.2	Test Symmetry property	131
17.1.4.3	Test Equivalence property	132
17.1.4.4	Test Inverse property	132
17.1.5	QuickCheck	132
17.1.5.1	Manual automation with QuickCheck properties	132
17.2	Write tests algorithm	133
17.3	Shrinking	133
<b>18</b>	<b>Logic</b>	<b>134</b>
18.1	Proposition	134
18.1.1	*	134
18.1.2	Atomic proposition	134
18.1.2.1	*	134
18.1.3	Compound proposition	134
18.1.3.1	*	134
18.1.4	Propositional logic	134
18.1.4.1	*	135
18.1.4.2	First-order logic	135
18.2	Logical connective	135
18.2.1	*	135
18.2.2	Conjunction	135
18.2.3	Disjunction	136
18.3	Predicate	136
18.4	Statement	136
18.4.1	*	136
18.5	Iff	137

<b>19 Haskell structure</b>	<b>138</b>
19.1 *	138
19.2 Pattern match	138
19.2.1 As-pattern	138
19.2.1.1 *	138
19.2.2 Wild-card	138
19.2.2.1 *	139
19.2.3 Case	139
19.2.4 Guard	139
19.2.4.1 *	139
19.2.5 Pattern guard	139
19.2.5.1 *	140
19.2.6 Lazy pattern	140
19.2.6.1 *	140
19.2.7 Pattern binding	140
19.2.7.1 *	141
19.3 Smart constructor	141
19.4 Level of code	141
19.4.1 *	141
19.4.2 Type level	141
19.4.2.1 Type level declaration	141
19.4.2.2 Type check	141
19.4.3 Term level	142
19.4.4 Compile level	142
19.4.4.1 *	142
19.4.5 Runtime level	142
19.4.6 Kind level	142
19.4.6.1 Kind check	142
19.5 Orphan type instance	143
19.6 undefined	143
19.7 Hierarchical module name	143
19.7.1 *	148
19.8 Reserved word	148
19.8.1 *	149
19.8.2 import	149
19.8.3 let	149
19.8.3.1 *	150
19.8.4 where	150
19.8.4.1 *	150
19.9 Haskell Language Report	150
19.9.1 *	150
19.10 Haskell'	150
19.10.1 *	151
19.11 Lense	151
<b>20 Computer science</b>	<b>152</b>
20.1 Guerrilla patch	152
20.1.1 Monkey patch	152
20.2 Interface	152
20.3 Module	152
20.4 Scope	152
20.4.1 Dynamic scope	152
20.4.2 Lexical scope	152
20.4.2.1 *	153
20.4.3 Local scope	153

20.4.3.1 *	153
20.5 Shadowing	153
20.6 Syntactic sugar	153
20.7 System F	153
20.7.1 *	153
20.8 Tail call	153
20.9 Thunk	153
20.10 Application memory	154
20.11 Turing machine	154
20.11.1 Turing complete	154
20.11.1.1 *	154
20.12 REPL	154
20.13 Domain specific language	154
20.13.1 *	154
20.13.2 Embedded domain specific language	155
20.13.2.1 *	155
20.14 Data structure	155
20.14.1 Cons cell	155
20.14.2 Construct	155
20.14.2.1 *	155
20.14.3 Leaf	155
20.14.4 Node	155
20.14.5 Spine	155
<b>21 Graph theory</b>	<b>157</b>
21.1 Successor	157
21.1.1 Direct successor	157
21.2 Predecessor	157
21.2.1 Direct predecessor	157
21.3 Degree	157
21.3.1 Indegree	157
21.3.2 Outdegree	157
21.4 Adjacency matrix	157
21.4.0.1 InstanceSigs	158
21.5 Strongly connected	158
21.5.1 *	158
21.5.2 Strongly connected component	158
21.5.2.1 *	158
<b>22 Tagless-final</b>	<b>159</b>
<b>III Give definitions</b>	<b>160</b>
<b>23 Identity type</b>	<b>161</b>
<b>24 Constant type</b>	<b>162</b>
<b>25 Gen</b>	<b>163</b>
<b>26 Tensorial strength</b>	<b>164</b>
<b>27 Strong monad</b>	<b>165</b>
<b>28 Weak head normal form</b>	<b>166</b>
28.1 *	166

<b>29 Function image</b>	<b>167</b>
29.1 * . . . . .	167
<b>30 Invertible</b>	<b>168</b>
<b>31 Invertibility</b>	<b>169</b>
<b>32 Define LANGUAGE pragma options</b>	<b>170</b>
32.1 ExistentialQuantification . . . . .	170
32.2 GADTs . . . . .	170
32.3 * . . . . .	170
32.4 GeneralizedNewTypeClasses . . . . .	170
32.5 FuncitonalDependencies . . . . .	170
<b>33 GHC check keys</b>	<b>171</b>
33.1 -Wno-partial-type-signatures . . . . .	171
<b>34 Generalised algebraic data types</b>	<b>172</b>
34.1 * . . . . .	172
<b>35 Order theory</b>	<b>173</b>
35.1 Domain theory . . . . .	173
35.2 Lattice . . . . .	173
35.3 Order . . . . .	173
35.3.1 Preorder . . . . .	173
35.3.1.1 * . . . . .	173
35.3.1.2 Total preorder . . . . .	173
35.3.2 Partial order . . . . .	174
35.3.2.1 * . . . . .	174
35.4 Partial order . . . . .	174
35.5 Total order . . . . .	174
<b>36 Universal algebra</b>	<b>175</b>
<b>37 Relation</b>	<b>176</b>
37.1 Reflexivity . . . . .	176
37.1.1 * . . . . .	176
37.2 Irreflexivity . . . . .	176
37.2.1 * . . . . .	176
37.3 Transitivity . . . . .	176
37.3.1 * . . . . .	177
37.4 Symmetry . . . . .	177
37.4.1 * . . . . .	177
37.5 Equivalence . . . . .	177
37.5.1 * . . . . .	177
37.6 Antisymmetry . . . . .	177
37.6.1 * . . . . .	177
37.7 Asymmetry . . . . .	177
37.7.1 * . . . . .	178
<b>38 Cryptomorphism</b>	<b>179</b>
38.1 * . . . . .	179
<b>39 Lexically scoped type variables</b>	<b>180</b>
<b>40 Abstract data type</b>	<b>181</b>
40.1 * . . . . .	181

41 Functional dependencies	182
42 MonoLocalBinds	183
43 KindSignatures	184
44 ExplicitNamespaces	185
45 Combinator pattern	186
46 Symbolic expression	187
46.1 * . . . . .	187
47 Polynomial	188
47.1 * . . . . .	188
48 Data family	189
49 Type synonym family	190
50 Indexed type family	191
50.1 * . . . . .	191
51 TypeFamilies	192
52 Error	193
52.1 * . . . . .	193
53 Exception	194
53.1 * . . . . .	194
54 ConstraintKinds	195
55 Specialisation	196
55.1 * . . . . .	196
56 Diagram	197
57 Cathegory theoretical presheaf	198
58 Topological presheaf	199
59 Diagonal functor	200
60 Limit functor	201
61 Dual vector space	202
62 Fundamental group	203
63 Algebra of continuous function	204
64 Tangent and cotangent bundle	205
65 Group action / representation	206
66 Lie algebra	207
67 Tensor product	208

68 Forgetful functor	209
69 Free functor	210
70 Homomorphism group	211
71 Representable functor	212
72 Corecursion	213
73 Coinduction	214
74 Initial algebra of an endofunctor	215
75 Terminal coalgebra for an endofunctor	216
 IV Citations	 217
 V Good code	 219
76 Good: Type aliasing	220
77 Good: Type wideness	221
78 Good: Print	222
79 Good: Fold	223
80 Good: Computation model	224
81 Good: Make bottoms only local	225
82 Good: Newtype wrap is ideally transparent for compiler and does not change performance	226
83 Good: Instances of types/type classes must go with code you write	227
84 Good: Functions can be abstracted as arguments	228
85 Good: Infix operators can be bind to arguments	229
86 Good: Arbitrary	230
87 Good: Principle of Separation of concerns	231
88 Good: Function composition	232
89 Good: Point-free	233
89.1 Good: Point-free is great in multi-dimensions . . . . .	233
90 Good: Functor application	234
91 Good: Parameter order	235
92 Good: Applicative monoid	236

<b>93 Good: Creative process</b>	<b>237</b>
93.1 Pick philosophy principles one to three the more - the harder the implementation	237
93.2 Draw the most blurred representation	237
93.3 Deduce abstractions and write remotely what they are	237
93.4 Model of computation	237
93.4.1 Model the domain	237
93.4.2 Model the types	237
93.4.3 Think how to write computations	237
93.5 Create	237
<b>94 «&lt;Good: About operators (&lt;\$ ) ( **&gt;) (&lt;* ) (&gt; ) »&gt;</b>	<b>238</b>
<b>95 Good: About functions like {mapM, sequence}_</b>	<b>239</b>
<b>96 Good: Guideliles</b>	<b>240</b>
96.1 Wiki.haskell	240
96.1.1 Documentation	240
96.1.1.1 Comments write in application terms, not technical.	240
96.1.1.2 Tell what code needs to do not how it does.	240
96.1.2 Haddock	240
96.1.2.1 Put haddock comments to ever exposed data type and function.	240
96.1.2.2 Haddock header	240
96.1.3 Code	240
96.1.3.1 Try to stay closer to portable (Haskell98) code	240
96.1.3.2 Try make lines no longer 80 chars	240
96.1.3.3 Last char in file should be newline	240
96.1.3.4 Symbolic infix identifiers is only library writer right	240
96.1.3.5 Every function does one thing.	240
<b>97 Good: Use Typed holes to progress the code</b>	<b>241</b>
<b>98 Good: Haskell allows infinite terms but not infinite types</b>	<b>242</b>
<b>99 Good: Use type sysonims to differ the information</b>	<b>243</b>
<b>100«&lt;Good: Control.Monad.Error -&gt; Control.Monad.Except»&gt;</b>	<b>244</b>
<b>101Good: Monad OR Applicative</b>	<b>245</b>
101.0.1 Start writing monad using 'return', 'ap', 'liftM', 'liftM2', '»' instead of 'do', '»='	245
101.0.2 Basic case when Applicative can be used	245
101.0.3 Applicative block vs Monad block	245
<b>102Good: Haskell Package Versioning Policy</b>	<b>246</b>
102.1 *	246
<b>103Good: Linear type</b>	<b>247</b>
<b>104Good: Exception vs Error</b>	<b>248</b>
<b>105Good: Let vs. Where</b>	<b>249</b>
<b>106Good: RankNTypes</b>	<b>250</b>
<b>107Good: Orphan type instance</b>	<b>251</b>
<b>108Good: Smart constructor</b>	<b>252</b>



109	Good: Thin category	253
110	Good: Recursion	254
111	Good: Monoid	255
112	Good: Free monad	256
113	Good: Use mostly where clauses	257
114	Good: Where clause is in a scope with function parameters	258
115	Good: Strong preference towards pattern matching over {head, tail, etc.} functions	259
116	Good: Patternmatching is possible on monadic bind in do	260
117	Good: Applicative vs Monad	261
118	Good: StateT, ReaderT, WriterT	262
119	Good: Working with MonadTrans and lift	263
120	Good: Don't mix Where and Let	264
121	Good: Where vs. Let	265
122	Good: The proper nature algorithm that models behaviour of many objects is computation heavy	266
123	Good: In Haskell parameters bound by lambda declaration instantiate to only one concrete type	267
124	Good: Instance is a good structure to draw a type line	268
125	Good: MTL vs. Transformers	269
VI	Bad code	270
126	Bad pragma	271
126.1	Bad: Dangerous LANGUAGE pragma option . . . . .	271
VII	Useful functions to remember	272
127	Prelude	273
127.1	Ord . . . . .	273
127.2	Calc . . . . .	273
127.3	List operations . . . . .	273
128	Data.List	274
129	Data.Char	275
130	QuickCheck	276

<b>VIII</b>	<b>Tools</b>	<b>277</b>
<b>131</b>	<b>ghc-pkg</b>	<b>278</b>
<b>132</b>	<b>Search over the Haskell packages code: Codesearch from Aelve</b>	<b>279</b>
<b>133</b>	<b>Integration of NixOS/Nix with Haskell IDE Engine (HIE) and Emacs (Spacemacs)</b>	<b>280</b>
133.11.	Install the Cachix . . . . .	280
133.22.	Installation of HIE . . . . .	280
133.2.12.1.	Provide cached builds . . . . .	280
133.2.22.2.a.	Installation on NixOS distribution: . . . . .	280
133.2.32.2.b.	Installation with Nix package manager: . . . . .	281
133.33.	Emacs (Spacemacs) configuration: . . . . .	281
133.44.	Open the Haskell file from a project . . . . .	282
133.55.	Be pleased writing code . . . . .	282
133.66.	(optional) Debugging . . . . .	282
<b>134</b>	<b>Debugger</b>	<b>284</b>
<b>135</b>	<b>GHCID</b>	<b>285</b>
<b>136</b>	<b>Continuous integration platrorms (CIs) for Open Source Haskell projets</b>	<b>286</b>
<b>IX</b>	<b>Libs</b>	<b>288</b>
<b>137</b>	<b>Exceptions</b>	<b>289</b>
137.1	Exceptions - optionally pure extensible exceptions that are compatible with the mtl	289
137.2	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. . . . .	289
137.3	Enclosed-exceptions - capture exceptions from the enclosed computation, while reacting to asynchronous exceptions aimed at the calling thread. . . . .	289
<b>138</b>	<b>Memory management</b>	<b>290</b>
138.1	membrain - type-safe memory units . . . . .	290
<b>139</b>	<b>Parsers - megaparsec</b>	<b>291</b>
<b>140</b>	<b>CLIs - optparse-applicative</b>	<b>292</b>
<b>141</b>	<b>HTML - Lucid</b>	<b>293</b>
<b>142</b>	<b>Web applications - Servant</b>	<b>294</b>
<b>143</b>	<b>IO libraries</b>	<b>295</b>
143.1	Conduit - practical, monolythic, guarantees termination return . . . . .	295
143.2	Pipes + Pipes Parse - modular, more primitive, theoretically driven . . . . .	295
<b>144</b>	<b>JSON - aeson</b>	<b>296</b>
<b>145</b>	<b>Backpack</b>	<b>297</b>
<b>X</b>	<b>Drafts</b>	<b>298</b>
<b>146</b>	<b>Exception handling</b>	<b>299</b>

<b>147</b>	<b>Constraints</b>	<b>302</b>
<b>148</b>	<b>Monad transformers and their type classes</b>	<b>304</b>
<b>149</b>	<b>Layering monad transformers</b>	<b>305</b>
<b>150</b>	<b>Hoogle</b>	<b>306</b>
150.1	Search . . . . .	306
150.2	Scope . . . . .	306
150.2.1	Default . . . . .	306
150.2.2	Hierarchical module name system (from big letter): . . . . .	306
150.2.3	Packages (lower case): . . . . .	307
<b>151</b>	<b>ST-Trick monad</b>	<b>308</b>
151.1 *	. . . . .	308
<b>152</b>	<b>Either</b>	<b>309</b>
152.1 *	. . . . .	309
<b>153</b>	<b>Inverse</b>	<b>310</b>
<b>154</b>	<b>Inversion</b>	<b>311</b>
<b>155</b>	<b>Inverse function</b>	<b>312</b>
<b>156</b>	<b>Inverse morphism</b>	<b>313</b>
<b>157</b>	<b>Partial inverse</b>	<b>314</b>
<b>158</b>	<b>PatternSynonyms</b>	<b>315</b>
158.1 *	. . . . .	315
<b>159</b>	<b>GHC debug keys</b>	<b>316</b>
159.1	ddump-ds . . . . .	316
159.1.1 *	. . . . .	316
<b>160</b>	<b>GHC optimize keys</b>	<b>317</b>
160.1	foptimal-applicative-do . . . . .	317
<b>161</b>	<b>Computational trinitarianism</b>	<b>318</b>
161.1 *	. . . . .	319
<b>162</b>	<b>Techniques functional programming deals with the state</b>	<b>320</b>
162.1	Minimizing . . . . .	320
162.2	Concentrating . . . . .	320
162.3	Deferring . . . . .	320
<b>163</b>	<b>Monadic Error handling</b>	<b>321</b>
<b>164</b>	<b>Functions</b>	<b>322</b>
<b>165</b>	<b>Void</b>	<b>323</b>
165.1 *	. . . . .	323
<b>166</b>	<b>Constructive proof</b>	<b>324</b>
<b>167</b>	<b>Intuitionistic logic</b>	<b>325</b>
167.1 *	. . . . .	325

<b>168</b>	<b>Principle of explosion</b>	<b>326</b>
168.1 *	.....	326
<b>169</b>	<b>Universal property</b>	<b>327</b>
<b>170</b>	<b>Yoneda lemma</b>	<b>328</b>
<b>171</b>	<b>Monoidal category, functoriality of ADTs, Profunctors</b>	<b>329</b>
<b>172</b>	<b>Const functor</b>	<b>330</b>
<b>173</b>	<b>Arrow in Haskell</b>	<b>331</b>
<b>174</b>	<b>Contravariant functor</b>	<b>332</b>
<b>175</b>	<b>Profunctor</b>	<b>333</b>
<b>XI</b>	<b>Reference</b>	<b>334</b>
<b>176</b>	<b>Functor-Applicative-Monad Proposal</b>	<b>335</b>
176.1 *	.....	335
<b>177</b>	<b>Haskell-98</b>	<b>336</b>
177.1	Old instance termination rules .....	336
<b>178</b>	<b>Performance results and comparisons of types &amp; solutions</b>	<b>337</b>
<b>XII</b>	<b>Liturgy</b>	<b>338</b>

```

      _oo0oo_
      o88888o
      88"  . "88
      (|  _-  |)
      0\   =  /0
      ---/\---'\-----
      . '  \ |      | //  ' .
      /  / | | | :   | | | \ \
      |  _ | | | | -:- | | | | \
      /   | \ \   -   ///  | \  \
      |   \ _ | ' '\---/' '   | /  |
      \   .-\_ _  ' '   ---/- .  /
      ---' . ' /---.-\ ` . ' .---
      . "" ' < ` .---\ < | > /---' > ' "" .
      | | : ` - \ ` ; \ _ / ; . / - ` : | |
      \ \ ` .   \ _ _ \ / _ _ / .- / /
      =====` .---` .--- \-----/-----' .-=====
      `-----'

```

# 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 Web book, PDF, L<sup>A</sup>T<sub>E</sub>X, or use Org-mode capable viewer/editor.

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

Users of Emacs can prettify radio targets to see as simple hyper-links with this Emacs snippet:

```
;;; 2019-06-12: NOTE:
;;; Prettify '<<Radio targets>>>' to be shown as '_Radio_targets_',
;;; when `org-descriptive-links` set.
;;; This is improvement of the code from: Tobias@glmorous:
;;; https://emacs.stackexchange.com/questions/19230/how-to-hide-targets
;;; There exists library created from the sample:
;;; https://github.com/talwrii/org-hide-targets
(defun custom-org-hidden-links-additional-re "\\(<<<\\)[[:print:]]+?\\(>>>\\)"
```

---

```
"Regular expression that matches strings where the invisible-property
  of the sub-matches 1 and 2 is set to org-link."
: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

[U+0627] [U+0644] [U+062C] [U+0628] [U+0631] *al-jabr* assemble parts

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

—

Additional meanings:

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

### 1.1 \*

Algebras

### 1.2 Algebraic

Composite from simple parts.

Also: [Algebraic data type](#).

### 1.3 Algebraic structure

\* includes axioms that must be satisfied and operations on the underlying (or "carrier") [set](#).

An underlying [set](#) with \* on top of it also called "an [algebra](#)".

\* 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 \* can be many modules, [algebras](#) and other vector spaces, and any variations that the definition includes.

Table 1.1: Algebraic structures

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

### 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

# 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.

## 2.1 \*

Category  
Categories

## 2.2 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 a stable [structure](#); for example it is regular and 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.

### 2.2.1 $*$

Abelian categories

## 2.3 Composition

Axiom of [Category](#).

### 2.3.1 \*

Composable  
Compositions

## 2.4 Endofunctor category

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

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

## 2.5 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 [endo-functors](#) 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.

### 2.5.1 `*`

Functors

### 2.5.2 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](#).

#### 2.5.2.1 `*`

`fmap`

Functor type class

#### 2.5.2.2 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 **fmap**ped and then **composed**.

### 2.5.2.3 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

### 2.5.2.4 Power set functor is a free monad

Since:

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

## 2.5.3 Functorial

Corresponds to **functor laws**.

### 2.5.4 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.

#### 2.5.4.1 \*

Forgetful

### 2.5.5 Identity functor

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

Denotation:

$1^{C \rightarrow C}$



### 2.5.6 Endofunctor

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

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

#### 2.5.6.1 \*

Endofunctors

### 2.5.7 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 form [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
```

#### 2.5.7.1 \*

[Applicative](#)

[Applicatives](#)

[Applicative functors](#)

**2.5.7.2 Applicative law****2.5.7.3 \***

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
```

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

- 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.

### 2.5.7.5 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`.

### 2.5.7.6 Monad

$\mu\acute{\nu}\omicron\varsigma$  *monos* sole  
 $\mu\omicron\nu\acute{\nu}\delta\alpha$  *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  $\{\text{fmap}, \text{pure}, (>>=)\}$ .

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 2.1: [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
Generates 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 a 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. ( $\gg$ )
- b.  $\gg=$
- c. ( $=\ll$ )
- d.  $=\ll$

- d. Sequencing operator ( $\gg$ ) == ( $*\gg$ ): 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**  $\mathcal{C}^{op}$ .

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 C`, `[U+2329]E,  $\overrightarrow{\eta}, \overrightarrow{\mu}$  [U+232A] monad` over `C`.

`Kleisli category  $\mathcal{C}_T$`  of `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 though `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](#).

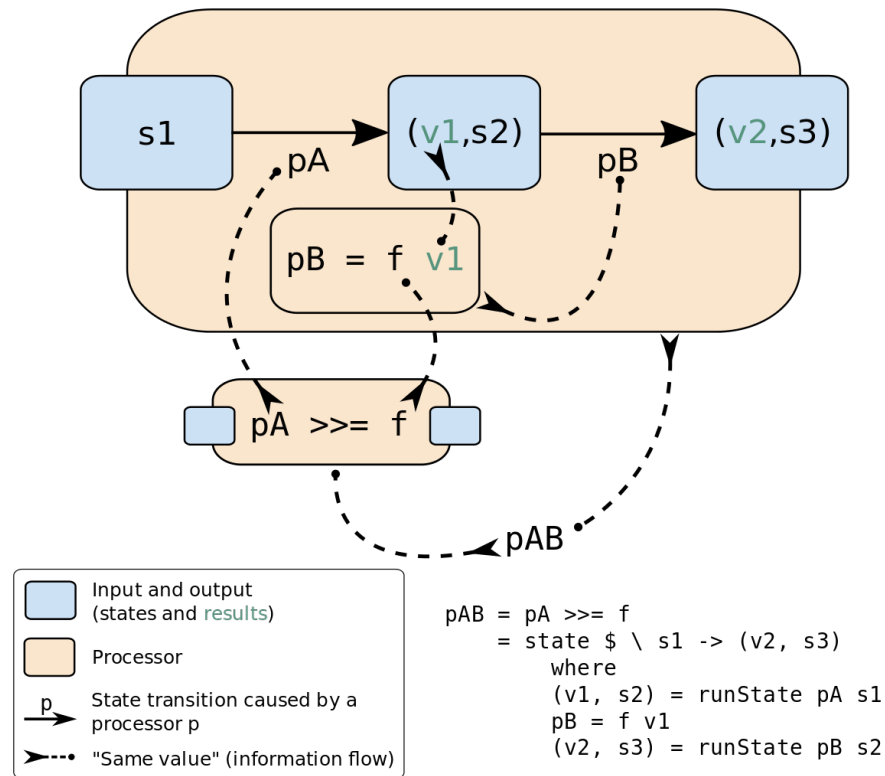
```
type State s = StateT s Identity
```

[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**  $m \rightarrow T \ m \rightarrow a$  arises, which does not allow to **unit**, **join** the  $m$  **monadic** layers.

$*$  have desirable properties and can add them to **monads**.  $*$  user 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 always 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

```

### 2.5.7.7 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 `(<|>) :: Type -> Type -> Type`.

Minimal complete definition:

```

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

```

a. `*`

Alternative

### 2.5.7.8 «<=\*>=>>

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

```

* > ≡ >>

```

### 2.5.8 Monoidal functor

Functors between monoidal categories that preserves monoidal structure.

### 2.5.9 Fusion

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

\* - functor axiom that allows to greatly simplify computations.

### 2.5.10 $\ll \leq \$ \geq \gg$

Get & set a value inside Functor.

### 2.5.11 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
```

#### 2.5.11.1 \*

Bifunctor

#### 2.5.12 \*

$\ll \leq \$ \geq \gg$

## 2.6 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.

### 2.6.1 \*

Hask

## 2.7 Magma

Set with a binary operation which form a closure.

### 2.7.1 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).

#### 2.7.1.1 \*

MAG

Magma category

Category of magmas

### 2.7.2 Semigroup

Magma with associative property of operation.

Defined in Haskell as:

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

**2.7.2.1 \***

Semigroups

**2.7.2.2 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

## 2.8 Morphism

$\mu \circ \rho \circ \varphi'$  *morphe* form

[Arrow](#) between two [objects](#) inside a [category](#).

[Morphism](#) can be anything.

[Morphism](#) is a generalization ( $f(x * y) \equiv f(x) \diamond f(y)$ ) of [homomorphism](#) ( $f(x * y) \equiv f(x) * f(y)$ ). Since general [morphisms](#) not so much often ment and discussed - under [morphism](#) people almost always really mean the meaning of [homomorphism](#)-like [properties](#), hense they discuss the [algebraic structures \(types\)](#) and homomorphisms between them.

In most usage, on a level under under the [objects](#): \* is most often means a map ([relation](#)) that translates from one mathematical [structure](#) (that source [object](#) represents) to another (that target [object](#) represents) (that is called (somewhat, somehow) "[structure](#)-preserving", but that [phrase](#) still means that translation can be lossy and irrevertable, so it is only bear reassemblence of preservation), and in the end the [morphism](#) can be anything and not hold to this conditions.

[Morphism](#) needs to correspond to [function](#) requirements to be it.

**2.8.1 \***

Morphisms

Arrow

Arrows

**2.8.2 Homomorphism** $\mu\acute{\varsigma}$  *homos* same (was chosen because of initial English mistranslation to "similar") $\mu\omicron\rho\varphi'$  *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](#).**2.8.2.1 \***

Homomorphic

**2.8.3 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](#).**2.8.3.1 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

### 2.8.3.2 Identity function

Return itself.

$(\backslash \ x.x)$

`id :: a -> a`

## 2.8.4 Monomorphism

$\mu o v o$  *mono* only

$\mu o r \varphi'$  *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  $f$ .

### 2.8.4.1 $*$

Monomorphic

Monomorphisms

## 2.8.5 Epimorphism

$\varepsilon \pi \iota$  *epi* on, over

$\mu o r \varphi'$  *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](#)

$g1 \circ f_{epi} = g2 \circ f_{epi} \Rightarrow g1 \equiv g2$  - 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.

### 2.8.5.1 $*$

Epimorphic

Epimorphisms

### 2.8.6 Isomorphism

$\sigma\omicron\varsigma$  *isos* equal  
 $\mu\omicron\rho\varphi'$  *morphe* form

Not equal, but equal for current intents and purposes.

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

Almost equal, but not quite:  $(\text{Integer}, \text{Bool})$  &  $(\text{Bool}, \text{Integer})$  - but can be transformed losslessly into one another.

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

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

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  $R \circ L = ID$

#### 2.8.6.1 \*

Isomorphic  
 Isomorphisms

#### 2.8.6.2 Lax

Holds up to [isomorphism](#).  
 (upon the transformation can be used as the same)

### 2.8.7 Endomorphism

$\varepsilon\nu\delta\omicron$  *endo* internal  
 $\mu\omicron\rho\varphi'$  *morphe* form

[Arrow](#) from [object](#) to itself.

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

#### 2.8.7.1 Automorphism

$\alpha\nu\tau\omicron$  *auto* self  
 $\mu\omicron\rho\varphi'$  *form* form

[Isomorphic endomorphism](#).

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



*a.* \*

Automorphic  
Automorphisms

### 2.8.7.2 \*

Endomorphic  
Endomorphisms

## 2.8.8 Catamorphism

$\kappa\alpha\tau$  *kata* downward  
 $\mu\omicron\rho\varphi$  *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](#).

### 2.8.8.1 \*

Catamorphic  
Catamorphisms

### 2.8.8.2 Catamorphism law

Table 2.2: [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.* Hylomorphism [catamorphism](#)  $\circ$  [anamorphism](#)

Expanding and collapsing the [structure](#).

*a.* \*

Hylomorphic  
Hylomorphisms

**2.8.8.3 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

**2.8.9 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:

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

**2.8.9.1 Kernel homomorphism**

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

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

**2.9 Set category**

[Category](#) in which [objects](#) are [sets](#), [morphisms](#) are [functions](#).

Denotation:

*Set*

**2.10 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.

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

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

Notice that the  $\overrightarrow{\eta}_{x^c}^{\mathcal{D}}((x, y)^c)$  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  $*$ :

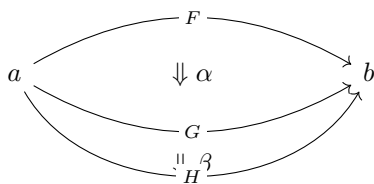
- for every object in  $\mathcal{C}$  picks natural transformation component in  $\mathcal{D}$ .
- for every morphism in  $\mathcal{C}$  picks the commuting diagram in  $\mathcal{D}$ , called naturality square.

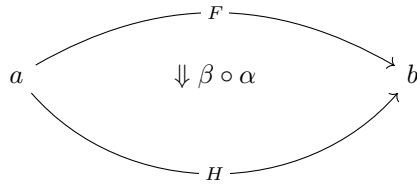
Also see: Natural transformation in Haskell

Knowledge of  $*$  forms a 2-category.

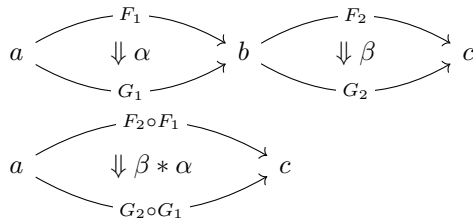
Can be composed

- "vertically":





- "horizontally" ("Godement [product](#)"):



Vertical and horizontal [compositions](#) can be done in any [order](#), they abide the exchange law.

### 2.10.1 \*

Natural transformations

Naturality condition

Naturality

### 2.10.2 Natural transformation component

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

#### 2.10.2.1 \*

Component of natural transformation

### 2.10.3 Natural transformation in Haskell

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

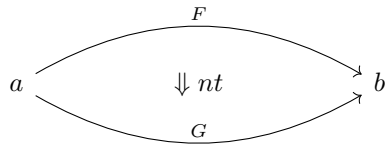
In [Hask](#) is  $F\ a \rightarrow G\ a$ . Can be analogued to repackaging data into another container, never modifies the [object](#) content, it only if - can delete it, because [operation](#) is lossy.

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

### 2.10.4 Cat category

[Category](#) where:

	Part	Is	#
*	<a href="#">object</a>	<a href="#">category</a>	0-cell
$\Rightarrow$	<a href="#">morphism</a>	<a href="#">functor</a>	1-cell
$\Rightarrow$	<a href="#">2-morphism</a>	<a href="#">natural transformation</a> , <a href="#">morphisms homotopy</a>	2-cell



Is Cartesian [closed category](#).

#### 2.10.4.1 \*

Cat

2-category

#### 2.10.4.2 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.

## 2.11 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

### 2.11.1 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.

## 2.12 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](#)).

### 2.12.1 \*

Proset category

Prosetal category

Poset category

Posetal category

## 2.13 Commuting diagram

Establishes equality in [morphisms](#) that have same source and target.

Draws the [morphisms](#) that are:

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

### 2.13.1 \*

Diagram commutes

Commutes

## 2.14 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.

### 2.14.1 \*

Universal constructions

## 2.15 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:  $\max \sum^{\forall} (! : c' \rightarrow c \mid p' = p \circ !, q' = q \circ !)$   
 $c'$  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](#).

### 2.15.1 \*

Products

## 2.16 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:  $\max \sum^{\forall} (! : c \rightarrow c' \mid i' = ! \circ i, j' = ! \circ j)$   
 $c'$  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](#).

### 2.16.1 \*

Coproducts

## 2.17 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.

## 2.18 Internal category

[Category](#) which is included into a bigger [category](#).

## 2.19 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.

### 2.19.1 \*

Hom-set

Hom sets

### 2.19.2 Hom-functor

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

[Functor](#) from the [product](#) of  $C$  with its [opposite category](#) to the [category](#) of [sets](#).

Denotation variants:

$$H_A = Hom(-, A)$$

$$h_A = C(-, A)$$

$$Hom(A, -) : C \rightarrow Set$$

Hom-[bifunctor](#):

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



### 2.19.3 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](#):

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

$a$  - [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.

#### 2.19.3.1 $*$

Function object

Internal hom

Exponential objects

Hom object

Hom objects

#### 2.19.3.2 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 3

## Data type

Set of values.

For [type](#) to have sense the values must share some sense, [properties](#).

### 3.1 \*

Type

Types

Data types

### 3.2 Actual type

[Data type](#) recieved by ->[inferring](#)->compiling->execution.

### 3.3 Algebraic data type

Composite [type](#) formed by combining other [types](#).

#### 3.3.1 \*

AlgDT

### 3.4 Cardinality

Number of possible implementations for a given [type](#) signature.

[Disjunction](#), sum - adds [cardinalities](#).

[Conjunction](#), [product](#) - multiplies [cardinalities](#).

#### 3.4.1 \*

Cardinalities

### 3.5 Data constant

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

### 3.6 Data constructor

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

### 3.7 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]
```

### 3.8 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](#).

#### 3.8.1 \*

Dependent types

### 3.9 Gen type

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

### 3.10 Higher-kinded data type

Any combination of \* and ->

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

*Humbly really a [function](#)*

#### 3.10.1 \*

Higher-kinded data types

## 3.11 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
```

## 3.12 Principal type

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

## 3.13 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
    }
```

### 3.13.1 \*

Product type

### 3.13.2 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 3.1: [Sequence constructor](#) naming by [arity](#)

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

### 3.13.2.1 \*

Sequences  
Tuples  
Ordered pair  
Ordered triple

### 3.13.2.2 List

The same [type objects](#) sequence.

Denotation:

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

Haskell definition:

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

Definition is self-referrential (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.

## 3.14 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`
```

## 3.15 Static typing

Typechecking takes place at compile level.

## 3.16 Structural type

Mathematical type. They form into structural type system.

### 3.16.1 \*

Structural

## 3.17 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.

### 3.17.1 \*

Structural typing

## 3.18 Sum data type

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

## 3.19 Type alias

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

## 3.20 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.

### 3.20.1 \*

Type classes

Typeclass

Typeclasses

### 3.20.2 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.

#### 3.20.2.1 Arbitrary function

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

### 3.20.3 CoArbitrary type class

Pseudogenerates a function basing on resulting `type`.

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

#### 3.20.3.1 \*

CoArbitrary

### 3.20.4 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`).

#### 3.20.4.1 \*

Typeable

### 3.20.5 Type class inheritance

Type class has a `superclass`.

### 3.20.6 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

#### 3.20.6.1 \*

Derived  
Deriving

## 3.21 Type constant

[Nullary type constructor](#).



## 3.22 Type constructor

Name of the [data type](#).

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

## 3.23 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](#).

## 3.24 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.

### 3.24.1 \*

Typed holes

## 3.25 Type inference

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

### 3.25.1 \*

Inferring  
Infer  
Infers  
Inferred

## 3.26 Type class instance

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

## 3.27 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](#).

### 3.27.1 \*

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

## 3.28 Type variable

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

## 3.29 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.

### 3.29.1 \*

Unlifted types

### 3.30 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.

#### 3.30.1 \*

Linear types

### 3.31 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

### 3.32 Session type

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

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

### 3.33 Binary tree

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

### 3.34 Bottom value

A `_` non-value in the type or pattern match expression. Placeholder for anything.

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

**3.34.1 \***

Bottom  
Bottom values

**3.35 Bound**

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

**3.35.1 \***

Bounded

**3.36 Constructor**

- a.* [Type constructor](#)
- b.* [Data constructor](#)

Also see: [Constant](#)

**3.36.1 \***

Constructors

**3.37 Context**

[Type constraints](#) for [polymorphic variables](#).  
Written before the main [type](#) signature, denoted:

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

**3.37.1 \***

Contexts

**3.38 Inhabit**

Values that is a component of [data type set](#).

## 3.39 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.

### 3.39.0.1 \*

Nodes

## 3.40 Expected type

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

## 3.41 ADT

*a.* `Abstract data type`

*b.* `Algebraic data type`

## 3.42 Concrete type

Fully resolved, definitive, non-`polymorphic type`.

## 3.43 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.

## 3.44 Kind

`Kind -> Type -> Data`

**3.44.1 \***

Kinds

**3.45 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 requestes 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 4

# Expression

Finite combination of symbols that is well-formed according to [context-free grammar](#).

Generally meaningless. Meaning gets [derived](#) from an  $*$  & [context](#) (and/or content words) by congruency with knowledge & experience.

### 4.1 $*$

Expressions

### 4.2 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.

### 4.3 RHS

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

### 4.4 LHS

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

## 4.5 Redex

Reducible expression.

## 4.6 Concatenate

Link together sequences, expressions.

## 4.7 Alpha equivalence

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

## 4.8 Ground expression

Expression that does not contain any free variables.

### 4.8.1 \*

Ground formula

## 4.9 Variable

A name for expression.

You probably would hear saying: Haskell has immutable "variables" (term used in imperative programming sense). Logically we see `statement` nonsensical in meaning, "variables" (something that has change as a defining property) are not changing, aka, nonsense. It is due to programmers tend to speak in terms of imperative programming. Read the saying as: Haskell has immutable values. In Haskell values (due to value semantics (see "Value")) are immutable, and Haskell expressions are functions (that are referentially transparent - meaning itself immutable) are also values (hence "functional programming" - functions are first-class values), and since values bind to \*, people are wrongly mix-up terms and say their names (according " \*") are immutable (inside scope area immutable value is binded in).

As you see in the code - Haskell variables (same names) hold different value at different time. Variables are reused, meaning "names are reused, binded to different values on scope changes/cycles, can be shadowed, and can have different values inside. But all values that Haskell holds - are, by the design of language, are treated immutable, any transformations Haskell resolves by creating new values, and frees the space by freeing-up from unneeded values.

### 4.9.1 \*

Variables



## 4.10 Phrase

Composable expression.

# Chapter 5

## 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  
\* -> \*

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

^ ^ ^ ^ ^ ^

\Name of the function  
\Parameter  
\Free variable  
\Bound variable  
\Var \Constant

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).

### 5.1 \*

Functions

Bound variable

## 5.2 Arity

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

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

## 5.3 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:

[U+2916]  
>->>  
 $f : X \text{ [U+2916] } Y$

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

Corresponds to [isomorphism](#).

### 5.3.1 \*

Bijjective  
Bijjective function

## 5.4 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.

### 5.4.1 $\Psi$ -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

--\
  * ---
--/

```

#### 5.4.1.1 \*

Psi-combinator

On-combinator

## 5.5 Function application

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

### 5.5.1 \*

Apply  
Applied  
Applying  
Application

## 5.6 Function body

[Expression](#) that haracterizes the [process](#).

## 5.7 Function composition

```

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

```

```

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

```

In Haskell inline [composition](#) requires:

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

### 5.7.1 \*

Composition

Compose

Composed

## 5.8 Function head

Is a part with name of the [function](#) and it's parameters.

AKA:  $f(x)$

## 5.9 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](#).

## 5.10 Higher-order function

[Function](#) that has [arity](#)  $> 1$ .

—

[HOF](#) is:

- [function](#) that accepts [function](#) as a [parameter](#)
- [function](#) that has more than one [parameter](#).

[Application](#) of an [argument](#) to [\\*](#) produces a [function](#) that has [arity](#)  $- 1$ .

### 5.10.1 \*

HOF

### 5.10.2 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.

## 5.11 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:

$\mapsto$

$\rightarrow$

$f : X \mapsto Y$

$f : X \rightarrow Y$

Corresponds to **Monomorphism**.

### 5.11.1 \*

Injective  
Injective function  
Injectivity

## 5.12 Partial function

One that does not cover all **domain**.  
**Unsafe** and causes trouble.

## 5.13 Purity

\* means properly abstracted.

If the contrary - **abstraction** is unpure.

Also see: **pure function**.

### 5.13.1 \*

Pure

## 5.14 Pure function

**Function** that is **pure**  $\equiv$  **referentially transparent function**.

## 5.15 Sectioning

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

## 5.16 Surjection

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

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

Denotation:

$$f : X \rightarrow Y$$

Corresponds to [Epimorphism](#).

### 5.16.1 \*

Surjective

Surjective function

## 5.17 Unsafe function

[Function](#) that does not cover at least one edge [case](#).

### 5.17.1 \*

Unsafe

## 5.18 Variadic

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

## 5.19 Domain

Source [set](#) of a [function](#).

$X$  in  $X \rightarrow Y$ .

## 5.20 Codomain

$Y$  in  $X \rightarrow Y$ .

[Codomain](#) - target [set](#) of a [function](#).

## 5.21 Open formula

Logical [function](#) that has [arity](#) and produces [proposition](#).

## 5.22 Recursion

Repeated [function application](#) when sometimes same [function](#) gets called.

Allows computation that may require indefinite amount of work.

### 5.22.1 \*

Recursive

### 5.22.2 Base case

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

### 5.22.3 Tail recursion

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

### 5.22.4 Polymorphic recursion

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

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

#### 5.22.4.1 \*

Milner–Mycroft typability  
Milner–Mycroft calculus

## 5.23 Free variable

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

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

## 5.24 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:

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

### 5.24.1 \*

Closed

## 5.25 Parameter

$\pi\alpha\rho'$  *para* subsidiary

$\mu'\tau\rho\upsilon$  *metron* measure

Named variable of a [function](#).

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

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

### 5.25.1 \*

Parameters

Formal parameter

Formal parameters

## 5.26 Partial application

Part of [function parameters](#) applied.

### 5.26.1 \*

Partially applied

## 5.27 Well-formed formula

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

### 5.27.1 \*

Well formed formula

WFF

wff

WFFS

wffs

## Chapter 6

# Homotopy

$\mu' \zeta$  homós same

One can be "continuously deformed" into the other.

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

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

### 6.1 \*

Homotopies

Homotopic

## Chapter 7

# Lambda calculus

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

\* has [lambda terms](#):

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

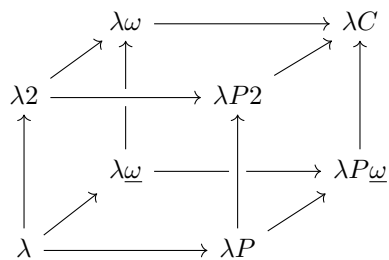
### 7.1 \*

Lambda term  
Lambda terms  
Lambda variable  
Lambda variables

### 7.2 Lambda cube

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

—



Each dimension of the cube corresponds to extensions (a new way of **objects** depending on **objects**):

- Polymorphic **types** - (**First-class polymorphism**) - terms allowed to depend on **types**, corresponding to **polymorphism**.
- **Type** operations - (**Type class**) - **types** depending on **types**, corresponding to **type** operators.
- Depended **types** - (**Higher-rank polymorphism**) - **types** depending on terms, corresponding to **dependent types**.

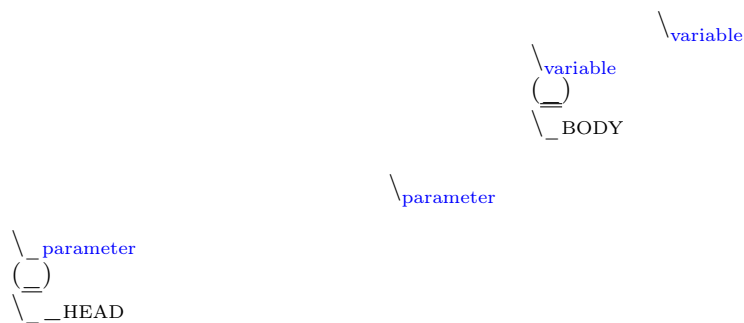
### 7.2.1 \*

$\lambda$ -cube  
 $\lambda$ -cube

## 7.3 Lambda function

**Function** of **Lambda calculus**.

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



### 7.3.1 \*

Lambda abstraction

### 7.3.2 Anonymous lambda function

**Lambda function** that is not binded to any name.

#### 7.3.2.1 \*

Anonymous lambda function

### 7.3.3 Uncurry

Replace sequenced lambda [functions](#) into single [function](#) taking [sequence/product](#) of values as [argument](#).

## 7.4 $\beta$ -reduction

Equation of a [parameter](#) to a [bound variable](#), then reducing [parameter](#) from the head.

### 7.4.1 \*

$\beta$  reduction  
Beta-reduction  
Beta reduction

### 7.4.2 $\beta$ -normal form

No [beta reduction](#) is possible.

#### 7.4.2.1 \*

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

## 7.5 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.

### 7.5.1 \*

«<CoC>»

## 7.6 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 7.1: 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> $a\ b$	$a / b$
$a \Rightarrow b$	$a \rightarrow b$	$b^a$

**7.6.1 \***

Curry–Howard isomorphism

Curry–Howard–Lambek

**7.7 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](#).

**7.7.1 \***

Curry

**7.8 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.

**7.8.1 \***

Hindley–Milner

Damas–Milner

Damas–Hindley–Milner

**7.9 Reduction**

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

See [Beta reduction](#)**7.9.1 \***

Reducible

## 7.10 $\beta$ - $\eta$ normal form

All  $\beta$ -reduction and  $\eta$ -abstraction are done in the expression.

### 7.10.1 \*

beta-eta normal form  
beta eta normal form

## 7.11 $\eta$ -abstraction

$$(\lambda x.Mx) \xrightarrow[\eta]{} M$$

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

### 7.11.1 \*

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

## 7.12 Lambda expression

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

# Chapter 8

## Operation

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

### 8.1 Constant

[Nullary operation](#).

Also see: [Type constant](#).

### 8.2 Binary operation

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

#### 8.2.1 \*

Binary operations

### 8.3 Operator

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

#### 8.3.1 Shift operator

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

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

##### 8.3.1.1 \*

Shift



### 8.3.2 Differential operator

Denotation.

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

Last one is partial.

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

#### 8.3.2.1 \*

Differential

## 8.4 Infix

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

For priorities see **Fixity**.

## 8.5 Fixity

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

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

Table 8.1: 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

#### 8.5.1 \*

Infixl

Infixr

Priority

Precedence

## 8.6 Zero

\* is the value with which [operation](#) always yields [Zero](#) value.

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

\* is distinct from [Identity](#) value.

## 8.7 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](#).

### 8.7.1 \*

Binds

Binding

Bindings

## 8.8 Declaration

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

## 8.9 Dispatch

Sort-out & send.

## 8.10 Evaluation

For FP see [Bind](#).

## Chapter 9

# Permutation

Bijjective function from domain to itself.

Domain & permutation functions & function composition form a group.

# Chapter 10

## 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
```

### 10.1 \*

Pointfree  
Tacit  
Tacit programming

### 10.2 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)
```

#### 10.2.1 \*

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

## 10.3 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
```

## 10.4 Squish

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

# Chapter 11

## Polymorphism

$\pi o\lambda' \varsigma$  *polús* many

At once several forms.

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

\* [types](#):

### 11.1 \*

Polymorphic

### 11.2 Levity polymorphism

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

### 11.3 Parametric polymorphism

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

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

[Types](#):

#### 11.3.1 Rank-1 polymorphism

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

##### 11.3.1.1 \*

Prenex

Prenex polymorphism

### 11.3.2 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](#).

### 11.3.3 Constrained polymorphism

Constrained [Parametric polymorphism](#).

#### 11.3.3.1 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

### 11.3.4 Impredicative polymorphism

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

#### 11.3.4.1 `*`

First-class polymorphism

### 11.3.5 Higher-rank polymorphism

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

There is a case where higher-rank [polymorphism](#) than the 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](#).

### 11.3.5.1 \*

Rank-n polymorphism

## 11.4 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.

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

## 11.5 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

```

## 11.6 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
```

## 11.7 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 12

## Pragma

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

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

### 12.1 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.

#### 12.1.1 LANGUAGE option

##### 12.1.1.1 \*

Language options

##### 12.1.1.2 Useful by default

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

##### 12.1.1.3 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.

##### 12.1.1.4 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`.

#### 12.1.1.5 ConstrainedClassMethods

Enable the definition of further `constraints` on individual class methods.

#### 12.1.1.6 CPP

Enable `C preprocessor`.

#### 12.1.1.7 DeriveFunctor

Automatic `deriving` of instances for the `Functor type class`.

For `type power set functor` is unique, its derivation implementation can be autochecked.

#### 12.1.1.8 ExplicitForAll

Allow explicit `forall` quantifier in places `where` it is implicit by Haskell.

#### 12.1.1.9 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`.

#### 12.1.1.10 FlexibleInstances

Allow `type class` instances `types` contain nested `types`.

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

Implies `TypeSynonymInstances`.

#### 12.1.1.11 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.)

#### 12.1.1.12 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.

#### 12.1.1.13 LambdaCase

Enables `expressions` of the form:

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

-- OR

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

#### 12.1.1.14 MultiParamTypeClasses

Implies: `ConstrainedClassMethods`

Enable the definitions of `typeclasses` with more than one `parameter`.

```
class Collection c a where
```

**12.1.1.15 MultiWayIf**

Enable multi-way-if syntax.

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

**12.1.1.16 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`.

**12.1.1.17 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.

**12.1.1.18 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`

#### 12.1.1.19 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`.

#### 12.1.1.20 TupleSections

Allow `tuple` section syntax:

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

#### 12.1.1.21 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)

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

#### 12.1.1.22 TypeSynonymInstances

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

**12.1.1.23 UndecidableInstances**

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

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

**12.1.1.24 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](#).

```
:: t1 -> t2) && (pattern t2)=
  then
    (ViewPattern (/expression/ -> /pattern/) :: t1) (return what originally was re-
    ceived into pattern match)
  else
    skip
```

- \* 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
```

**12.1.1.25 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.

**12.1.1.26 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

#### 12.1.1.27 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](#).



### 12.1.2 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 13

# 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](#).

### 13.1 \*

Principle of compositionality

Composition

Compositional

## Chapter 14

# 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**.

### 14.1 \*

Referentially transparent

# Chapter 15

## Semantics

Philosophical study of meaning.  
Meaning of symbols, words.

### 15.1 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](#).

#### 15.1.1 Argument

*arguere* to make clear, to shine

\* - evidence, proof, [statement](#) that results in system consequences.

##### 15.1.1.1 Argument of a function

A value binded to the [function parameter](#). Value/topic that the fuction would [process](#)/deal with.

Also see Argument.

*a.* \*

Function argument

##### 15.1.1.2 First-class

Means *it*:

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

From 1&2  $\rightarrow$  *it* can include itself.

### 15.1.2 Relation

[Relationship](#) between two [objects](#).

By default it is not directed and not limited.

In [Set theory](#): some subset of a [Cartesian product](#) between [sets](#) of [objects](#).

#### 15.1.2.1 \*

Relations

Relationship

## 15.2 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](#).

### 15.2.1 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](#)).

#### 15.2.1.1 \*

Abstractions

Abstracting

Abstract

#### 15.2.1.2 Leaky abstraction

[Abstraction](#) that leaks details that it is supposed to [abstract](#) away.

a. \*

Leaky abstractions

### 15.2.1.3 Object

Absolute [abstraction](#).

Point.

Can have [properties](#).

Often abstracts something, for example some [structure](#), [maybe](#) mathematical.

[Objects](#) without [process](#) are in [constant](#) state.

a. \*

Structure

Structures

Objects

b. Arrow Second level of absolute [abstraction](#).

[Arrow](#).

Can have target, can have source. Both often are [objects](#).

Often abstracts [process](#).

Can have [properties](#).

Also alias in [Category](#) Theory for "[morphism](#)", thou theory imposes [properties](#).

a. \* Arrows

Process

c. Terminal object One that receives 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 receives *terminal* or [unit arrow](#).

[Dual](#) of [initial object](#).

Denotation:

Category theory

1

Haskell

()

- d. Initial object One that emits unique **arrow** into every **object**.

$$\exists ! : \emptyset \rightarrow x \mid \exists \emptyset \in \mathcal{C}, \forall x \in \mathcal{C}$$

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**

- e. Value What **object** abstracts. Without any **object** external **structure** (aka **identity** in **Category Theory**). So **\*** is immutable. Such heresy is called "Value **semantics**" and leads such things as **referential transparency**, functional programming and Haskell.

(Except, when you hack Haskell with explicit low-level funtions, and start to directly mute values - then you are on your own, Haskell paradigm does not expect that.)

- a. \* Value **semantics**  
Values

- f. Tensor **Object** existing out of planes, thus it can translate **objects** from one plane into another.  
\* can be tried to be described with knowledge existing inside planes (from projection on the plane), but representation would always be partial.

**Tensor** of rank 1 is a vector.

Translations with **tensor** can be seen as **functors**.

- a. \*  
Tensors  
Tensorial

### 15.2.2 Ambigram

ambi both

γρ´μμα *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.*

### 15.2.3 Binary

Two of something.

### 15.2.4 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**.

### 15.2.5 Refutable

One that has an option to fail.

### 15.2.6 Irrefutable

One that can not fail.

### 15.2.7 Superclass

Broader parent class.

### 15.2.8 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 ()`.

### 15.2.9 Nullary

Takes no entries (for example has the `arity` of `zero`).  
Has the trivial `domain`.

### 15.2.10 Syntax tree

Tree of syntactic elements (each `node` denotes `construct` occurring in the language/source code) that represent the full particular `expression`/implementation (or said).

#### 15.2.10.1 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.

*a.* \*  
AST

#### 15.2.10.2 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.

*a.* \*  
Parse tree  
Derivation tree

### 15.2.11 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.

### 15.2.12 Linear

Values consumed once or not used.

$x^2$  consumes/uses  $x$  two times ( $x*x$ ).

#### 15.2.12.1 \*

Linearity

### 15.2.13 Predicative

Non-self-referencing definition.

—

*Antonym - [Impredicative](#).*

### 15.2.14 Quantifier

Specifies the quantity of specimens.

Two most common [quantifiers](#)  $\forall$  ([Forall](#)) and  $\exists$  (Exists).  
 $\exists!$  - one and only one (exists only unique).

#### 15.2.14.1 \*

Quantification  
 Quantifiers  
 Quantified

#### 15.2.14.2 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](#)

*a.* \*  
Forall

## 15.3 Axiomatic semantics

Empirical [process](#) of studying something complex by finding and analyzing true [statements](#) about it.

Good for examining interconnections.

### 15.3.1 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](#) which does what it says.

#### 15.3.1.1 \*

Properties

#### 15.3.1.2 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

### 15.3.1.3 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

### 15.3.1.4 Right associative

\* - the operations are grouped from the right.

### 15.3.1.5 Non-associative

Operations can't be chained.

Often is the [case](#) when the output [type](#) is incompatible with the input [type](#).

### 15.3.1.6 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

#### 15.3.1.7 Commutativity

$\forall(a, b) \in S : P(a, b) \equiv P(b, a)$

a. \*

Commutative

Commutative law

#### 15.3.1.8 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

#### 15.3.1.9 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

### 15.3.2 Effect

Observable action.

### 15.3.3 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.

#### 15.3.3.1 \*

Bisimilar

## 15.4 Content word

Words that name [objects](#) of reality and their qualities.

## 15.5 Ancient Greek and Latin prefixes

### 15.5.1 \*

Greek prefix  
Latin prefix

## 15.6 Idiom

\* - something having a meaning that can not be [derived](#) from the conjoined meanings of \* constituents.

Meaning can be special for language speakers or human with particular knowledge.

\* can also mean [applicative functor](#), people better stop making [idiom](#) from the term "[idiom](#)".

### 15.6.1 \*

Idioms  
Idiomatic

Table 15.1: 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)

## 15.7 Impredicative

Self-referencing definition.

—

*Antonym - [Predicative](#).*

## 15.8 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.

### 15.8.1 \*

CFG



# Chapter 16

## Set

Well-defined collection of distinct [objects](#).

### 16.1 \*

Sets  
Set theory

### 16.2 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.

### 16.3 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:  
 $\mathcal{P}(\mathcal{S})$

### 16.4 Singleton

[Singleton](#) - [unit set](#) - [set](#) with exactly one element.  
Also 1-[sequence](#).

### 16.5 Russell's paradox

If there exists normal [set](#) of all [sets](#) - it should contain itself, which makes it abnormal.

## 16.6 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**

### 16.6.1 Pullback

Subset of the **cartesian product** of two **sets**.

#### 16.6.1.1 \*

Pullbacks

# Chapter 17

## Testing

### 17.1 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](#).

#### 17.1.1 Function property

[Property](#) corresponds to the according law.

In [property testing](#) you need to think additionally about [generator](#) and [shrinking](#).

#### 17.1.2 Property testing type

Table 17.1: [Property testing types](#)

	Exhaustive	Randomized	<a href="#">Unit</a> test
Whole <a href="#">set</a> of values	Exhaustive <a href="#">property</a> test	Randomised <a href="#">property</a> test	One element
Special subset of values	Exhaustive specialised <a href="#">property</a> test	Randomised specialised <a href="#">property</a> test	One element

#### 17.1.3 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 counterexample, finds a part that still fails, and recurses [shrinking](#).

**17.1.3.1 \***

Generators

**17.1.3.2 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
```

**17.1.4 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)
```

**17.1.4.1 Test Commutative property**

[Commutativity](#)

```
:: Arbitrary a => (a -> a -> a) -> Property
```

**17.1.4.2 Test Symmetry property**

[Symmetry](#)

```
:: Arbitrary a => (a -> a -> Bool) -> Property
```

### 17.1.4.3 Test Equivalence property

Equivalence

```
:: (Arbitrary a, Eq b) => (a -> b) -> (a -> b) -> Property
```

### 17.1.4.4 Test Inverse property

```
:: (Arbitrary a, Eq b) => (a -> b) -> (b -> a) -> Property
```

## 17.1.5 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.

### 17.1.5.1 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
  quickCheck $ eq $ (prop_AutoFunctorCompose :: Prop_AutoFunctorCompose (Four' ())) Integer String
```

## 17.2 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.

## 17.3 Shrinking

**Process** of reducing complexity in the test **case** - re-run with smaller values and make sure that the test still fails.

# Chapter 18

## Logic

### 18.1 Proposition

Purely abstract & theoretical logical [object](#) (idea) that has a Boolean value.

\* is expressed by a [statement](#).

#### 18.1.1 \*

Propositions

#### 18.1.2 Atomic proposition

Logically undividable [unit](#). Does not contain [logical connectives](#).

##### 18.1.2.1 \*

Atomic propositions

#### 18.1.3 Compound proposition

Formed by connecting [propositions](#) by [logical connectives](#).

##### 18.1.3.1 \*

Compound propositions

#### 18.1.4 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.

**18.1.4.1 \***

Proposition logic  
 Proposition calculus  
 Propositional calculus  
 Statement logic  
 Sentential logic  
 Sentential calculus  
 Zeroth-order logic

**18.1.4.2 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.

**18.2 Logical connective**

Logical [operation](#).

**18.2.1 \***

Logical connectives

**18.2.2 Conjunction**

Logical AND.

Denotation:

$\wedge$

Multiplies [cardinalities](#).

Haskell [kind](#):



\* \*

### 18.2.3 Disjunction

Logical *OR*

Denotation:

$\vee$

Summs [cardinalities](#).

## 18.3 Predicate

[Function](#) with Boolean [codomain](#).

$P : X \rightarrow \{true, false\}$  - \* on  $X$ .

Notation:  $P(x)$

Im many cases includes [relations](#), [quantifiers](#).

## 18.4 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".

### 18.4.1 \*

Assertion

Assertions

Statements

## 18.5 Iff

If and only if, exactly when, just.

Denotation:

$\iff$

# Chapter 19

## Haskell structures

### 19.1 \*

Haskell structures

### 19.2 Pattern match

Are not first-class. It is a set of pattern match semantic notations.

Must be linear.

\* precedence (especially with more than one parameter, especially with \_ used) often changes the function.

#### 19.2.1 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
```

##### 19.2.1.1 \*

As-patterns

As pattern

As patterns

#### 19.2.2 Wild-card

Matches anything and can not be binded. For matching something that should pass not checked and is not used.

```
head (x:_) = x
tail (_:xs) = xs
```

**19.2.2.1 \***

Wild-cards  
 Wildcard  
 Wildcards

**19.2.3 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](#).

**19.2.4 Guard**

Check values against the [predicate](#) and use the first match definition:

```
f x
  | predicate1 = definition1
  | predicate2 = definition2
  ...
  | x < 0      = definitionN
  ...
  | otherwise  = definitionZ
```

**19.2.4.1 \***

Guards

**19.2.5 Pattern guard**

Allows check a [list](#) of pattern matches against [functions](#), and then proceed.

$$(patternMatch1) <- (funcCheck1)$$

```
, (patternMatch2) <- (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.

### 19.2.5.1 \*

Pattern guards

## 19.2.6 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.

### 19.2.6.1 \*

Lazy-pattern  
Lazy patterns

## 19.2.7 Pattern binding

Entire [LHS](#) is a pattern, is a [lazy pattern](#).

```
fib@(1:tfib) = 1 : 1 : [ a+b | (a,b) <- zip fib tfib ]
```

**19.2.7.1 \***

Pattern bindings

**19.3 Smart constructor**

[Process](#)/code placing extra rules & [constraints](#) on the construction of values.

**19.4 Level of code**

There are these levels of Haskell code:

**19.4.1 \***

Code level

**19.4.2 Type level**

[Level of code](#) that works with [data types](#).

**19.4.2.1 Type level declaration**

```
type ...
newtype ...
data ...
class ...
instance ...
```

*a.* \*

Type level declarations  
 Type-level declaration  
 Type-level declarations

**19.4.2.2 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 algorithym over/instead of [CUSK](#).

a. \*

CUSK  
CUSKs  
Complete user-specific kind signatures  
Complete, user-specific kind signature

### 19.4.3 Term level

[Level of code](#) that does logical execution.

### 19.4.4 Compile level

[Level of code](#), about compilation processes/results.

#### 19.4.4.1 \*

Compilation level

### 19.4.5 Runtime level

[Level of code](#) of main program [operation](#), when machine does computations with compiled [binary](#) code.

### 19.4.6 Kind level

[Level of code](#) where [kinds](#) & [kind](#) declarations are situated, infered and checked.

#### 19.4.6.1 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

## 19.5 Orphan type instance

Hanging [type](#) instance from inconsistent code base.

- a. Supporting [structure](#) not fully present.
- b. Several implementations of instance present.

## 19.6 undefined

Placeholder value that helps to do [typechecking](#).

## 19.7 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>

```

### 19.7.1 \*

Top-level module name  
 Top-level module names

## 19.8 Reserved word

Haskell has special meaning for:

`case`, `class`, `data`, `deriving`, `do`, `else`, `if`, `import`, `in`, `infix`, `infixl`, `infixr`, `instance`, `let`, `of`, `mo`

### 19.8.1 \*

Reserved words

### 19.8.2 `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.

### 19.8.3 `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
```

**19.8.3.1 \***

Let expression  
Let expressions

**19.8.4 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 ...
```

**19.8.4.1 \***

Where clause

**19.9 Haskell Language Report**

Document that is a standart of language.

**19.9.1 \***

Report  
Haskell Report  
Haskell 98 Language Report  
Haskell 98 Report  
Haskell 1998 Language Report  
Haskell 2010 Language Report  
Haskell 2010 Report

**19.10 Haskell'**

Current language development mod.

<https://prime.haskell.org/>

**19.10.1   \***

Haskell prime

**19.11   Lense**

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



# Chapter 20

## Computer science

### 20.1 Guerrilla patch

\* changing code/[applying](#) patch sneakily - and possibility incompatibility with other at runtime. [Monkey patch](#) is derivative term.

#### 20.1.1 Monkey patch

*From [Guerrilla patch](#).*

\* is a way for program to modify supporting system software affecting only the running instance of the program.

### 20.2 Interface

Point of mutual meeting. Code behind [interface](#) determines how data is consumed.

### 20.3 Module

Importable organizational [unit](#).

### 20.4 Scope

Area [where binds](#) are accessible.

#### 20.4.1 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](#).

#### 20.4.2 Lexical scope

[Scope bound](#) by the [structure](#) of source code [where](#) the named entity is defined.

**20.4.2.1 \***

Static scope

**20.4.3 Local scope**

[Scope](#) applies only in (current) area.

**20.4.3.1 \***

Local

**20.5 Shadowing**

When in the [local scope](#) bigger [scope variable](#) overridden by same name [variable](#) from the [local scope](#).

**20.6 Syntactic sugar**

Artificial way to make language easier to read and write.

**20.7 System F**

Is [parametric polymorphism](#) in programming.

Extends the [Lambda calculus](#) by introducing  $\forall$  (universal [quantifier](#)) over [types](#).

**20.7.1 \***

Girard–Reynolds polymorphic lambda calculus  
Girard-Raynolds

**20.8 Tail call**

Final [evaluation](#) inside the [function](#). Produces the [function](#) result.

**20.9 Thunk**

Not evaluated calculation. Can be dragged around, until be lazily evaluated.

Table 20.1: 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

## 20.10 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.

## 20.11 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.

### 20.11.1 Turing complete

[Set](#) of action rules that can simulate any [Turing machine](#).

#### 20.11.1.1 \*

Turing incomplete  
 Turing incompleteness  
 Turing completeness  
 Computationally universal

## 20.12 REPL

Read-eval-print loop, aka interactive shell.

## 20.13 Domain specific language

Language design/fitted for particular [domain](#) of [application](#). Mainly should be [Turing incomplete](#), since general-purpose language implies [Turing completeness](#).

### 20.13.1 \*

Domain-specific language  
 DSL

### 20.13.2 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.

#### 20.13.2.1 \*

eDSL

## 20.14 Data structure

### 20.14.1 Cons cell

Cell that values may inhabit.

### 20.14.2 Construct

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

#### 20.14.2.1 \*

Cons

### 20.14.3 Leaf

-

### 20.14.4 Node

\*  
/ \

### 20.14.5 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  []  
1:2:3: []  
Spine:  
  :  
 / \  
-  :  
  / \  
  - :  
    / \  
    - []
```

# Chapter 21

## Graph theory

### 21.1 Successor

**Object** that receives the **arrow**.

#### 21.1.1 Direct successor

Immediate **successor**.

### 21.2 Predecessor

**Object** that sends **arrow**.

#### 21.2.1 Direct predecessor

Immediate **predecessor**.

### 21.3 Degree

Number of **arrows** of **object**.

#### 21.3.1 Indegree

Number of ingoing **arrows**.

#### 21.3.2 Outdegree

Number of outgoing **arrows**.

### 21.4 Adjacency matrix

Matrix of connection of objects  $\{-1, 0, 1\}$ .

**21.4.0.1 InstanceSigs**

Allow adding [type](#) signatures to [type class function](#) instance [declaration](#).

**21.5 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 ( $\Theta(V+E)$ ).

[Binary relation](#) of being [strongly connected](#) is an [equivalence relation](#).

**21.5.1 \***

Strongly-connected

**21.5.2 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](#).

**21.5.2.1 \***

SCC

Strongly connected components

Strongly-connected component

Strongly-connected components

## Chapter 22

# 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 23

### Identity type

## Chapter 24

### Constant type

## Chapter 25

### Gen

## Chapter 26

# Tensorial strength

## Chapter 27

# Strong monad

## Chapter 28

# Weak head normal form

### 28.1 \* WHNF

## Chapter 29

# Function image

### 29.1 \*

Image



## Chapter 30

# Invertible

## Chapter 31

# Invertibility

## Chapter 32

# Define LANGUAGE pragma options

### 32.1 ExistentialQuantification

### 32.2 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](#).

### 32.3 \*

GADT

Generalized algebraic data type

First-class phantom data type

Guarded recursive data type

Equality-qualified data type

### 32.4 GeneralizedNewTypeClasses

### 32.5 FuncitonalDependencies

## Chapter 33

# GHC check keys

### 33.1 -Wno-partial-type-signatures

Supresses [PartialTypeSignatures wildcard infer](#) warning.

## Chapter 34

# Generalised algebraic data types

LANGUAGE [GADTs](#)

### 34.1 \*

GADT

# Chapter 35

## Order theory

Investigates in thepht the intuitive notion of [order](#) using [binary relations](#).

### 35.1 Domain theory

Formalizes approximation and convergense.  
Has close [relation](#) to Topology.

### 35.2 Lattice

[Abstract structure](#) that consists of [partially ordered set](#), [where](#) every two elements have unique supremum and infimum.  $\Rightarrow$  \* [algebraic structure](#) satisfying certain axiomatic identities.  
\* [order-theory](#) & [algebraic](#).

### 35.3 Order

#### 35.3.1 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](#).

##### 35.3.1.1 \*

Preordered

##### 35.3.1.2 Total preorder

$\forall a, b : a \leq b \vee b \leq a \Rightarrow \text{Total Preorder}.$

### 35.3.2 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.

#### 35.3.2.1 $*$

Partial orders

Partially ordered set

Partially ordered sets

Poset

Posets

## 35.4 Partial order

## 35.5 Total order

## Chapter 36

# Universal algebra

Studies [algebraic structures](#).



# Chapter 37

## Relation

### 37.1 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](#).

#### 37.1.1 \*

Reflexive

Reflexive relation

### 37.2 Irreflexivity

$$R^{X \rightarrow X}, \forall x \in X : \nexists R(x, x)$$

#### 37.2.1 \*

Anti-reflexive

Anti-reflexive relation

Irreflexive

Irreflexive relation

### 37.3 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.

**37.3.1 \***

Transitive  
Transitive relation

**37.4 Symmetry**

$$\forall a, b \in X : (aRb \iff bRa)$$

**37.4.1 \***

Symmetric  
Symmetric relation

**37.5 Equivalence**

Reflexive	Symmetric	Transitive
$\forall x \in X, \exists R : xRx$ $a = a$	$\forall a, b \in X : (aRb \iff bRa)$ $a = b \iff b = a$	$\forall a, b, c \in X, \forall R^{X \rightarrow X} : (aRb \wedge bRc) \Rightarrow aRc$ $a = b, b = c \Rightarrow a = c$

**37.5.1 \***

Equivalent  
Equivalent relation

**37.6 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.

**37.6.1 \***

Antisymmetric  
Antisymmetric relation

**37.7 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.

**37.7.1   \***

Asymmetric

Asymmetric relation

## Chapter 38

# Cryptomorphism

[Equivalent](#), interconvertable with no loss of information.

### 38.1 \*

Crypromorphic

## Chapter 39

# Lexically scoped type variables

Enable [lexical scope](#) for [forall quantifier](#) defined [type variables](#)

Implemented in [ScopedTypeVariables](#)

## Chapter 40

# 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**.

### 40.1 \*

AbsDT

## Chapter 41

# Functional dependencies

## Chapter 42

# MonoLocalBinds



## Chapter 43

# KindSignatures

## Chapter 44

# ExplicitNamespaces

## Chapter 45

# Combinator pattern

## Chapter 46

# 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  $*$ :  $(x \ y)$ .

### 46.1 $*$

S-expression

S-expressions

Sexpression

Sexpressions

Sexp

Sexprs

Sexpr

Sexprs

## Chapter 47

# Polynomial

[Expression](#) consisting of:

- [variables](#)
- coefficients
- addition
- subtraction
- multiplication (including positive integer [variable](#) exponentiation)

[Polynomials](#) form a [ring](#). [Polynomial ring](#).

### 47.1 \*

Polynomials

## Chapter 48

# Data family

Indexed form of data and newtype definitions.

## Chapter 49

# Type synonym family

Indexed form of [type](#) synonyms.

## Chapter 50

# 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](#).

## 50.1 \*

Type family



## Chapter 51

# 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 52

# Error

Mistake in the program that can be resolved only by fixing the program.

`error` is a sugar for `undefined`.

Distinct from [Exception](#).

### 52.1 \*

Errors

## Chapter 53

# Exception

Expected but irregular situation.

Distinct from [Error](#). Also see Exception vs Error

### 53.1 \*

Exceptions

## Chapter 54

# 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 55

# Specialisation

Turns [ad hoc polymorphic function](#) into compiled [type](#)-specific implementations.

### 55.1 \*

Specialise  
Specialize  
Specialization

## Chapter 56

# Diagram

For [categories](#)  $C$  and  $J$ , a [diagram](#) of [type](#)  $J$  in  $C$  is a [covariant functor](#)  $D : J \rightarrow C$ .

## Chapter 57

# Category theoretical presheaf

For categories  $C$  and  $J$ , a  $J$ -presheaf on  $C$  is a contravariant functor  $D : C \rightarrow J$ .

## Chapter 58

# 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 59

# 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 60

# 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 61

# 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 62

# 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  $\pi_1(X, x_0)$  to  $\pi_1(Y, y_0)$ . 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 63

# 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  $C(f)(\varphi) = \varphi \circ f$  for every  $\varphi$  in  $C(Y)$ .

## Chapter 64

# 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 65

# 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 66

# Lie algebra

Assigning to every real (complex) Lie [group](#) its real (complex) [Lie algebra](#) defines a [functor](#).



## Chapter 67

# 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 68

# 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 69

# 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 70

# 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  $\text{Abop} \times \text{Ab} \rightarrow \text{Ab}$  (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  $\varphi \mapsto g \circ \varphi \circ f$ . See Hom functor.

## Chapter 71

# 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](#)  $C^{\text{op}} \times C \rightarrow \text{Set}$ . 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  $\varphi \mapsto g \circ \varphi \circ f$ .

[Functors](#) like these are called representable [functors](#). An important goal in many settings is to determine whether a given [functor](#) is representable.

## Chapter 72

# Corecursion

## Chapter 73

# Coinduction

*proper definition*

\* [dual](#) to induction.

Generalises to [corecursion](#).

## Chapter 74

# Initial algebra of an endofunctor



## Chapter 75

# 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 76

# Good: Type aliasing

Use [data type](#) aliases to deferentiate logic of values.

## Chapter 77

# 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 78

### Good: Print

```
print :: Show a => a -> IO ()  
print a = putStrLn (show a)
```

## Chapter 79

# 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 80

# Good: Computation model

Model the [domain](#) and [types](#) before thinking about how to write computations.

## Chapter 81

Good: Make bottoms only local

## Chapter 82

Good: Newtype wrap is ideally transparent for compiler and does not change performance

## Chapter 83

Good: Instances of types/type classes must go with code you write

## Chapter 84

Good: Functions can be abstracted  
as arguments

## Chapter 85

Good: Infix operators can be bind  
to arguments

## Chapter 86

# 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 87

# Good: Principle of Separation of concerns



## Chapter 88

# 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 89

# 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.

### 89.1 Good: Point-free is great in multi-dimensions

BigData and OLAP analysis.

## Chapter 90

# 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 91

# 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 92

# 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 93

### Good: Creative process

- 93.1 Pick philosophy principles one to three the more - the harder the implementation
- 93.2 Draw the most blurred representation
- 93.3 Deduce **abstractions** and write remotely what they are
- 93.4 Model of computation
  - 93.4.1 Model the **domain**
  - 93.4.2 Model the **types**
  - 93.4.3 Think how to write computations
- 93.5 Create

## Chapter 94

«<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 95

Good: About functions like `{mapM, sequence}_`

Trailing `_` means ignoring the result.



# Chapter 96

## Good: Guideliles

### 96.1 Wiki.haskell

#### 96.1.1 Documentation

96.1.1.1 Comments write in **application** terms, not technical.

96.1.1.2 Tell what code needs to do not how it does.

#### 96.1.2 Haddock

96.1.2.1 Put haddock comments to ever exposed **data type** and **function**.

96.1.2.2 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>  
-}
```

#### 96.1.3 Code

96.1.3.1 Try to stay closer to portable (Haskell98) code

96.1.3.2 Try make lines no longer 80 chars

96.1.3.3 Last char in file should be newline

96.1.3.4 Symbolic **infix** identifiers is only library writer right

96.1.3.5 Every **function** does one thing.

## Chapter 97

# Good: Use Typed holes to progress the code

[Typed holes](#) help build code in complex situations.

## Chapter 98

# Good: Haskell allows infinite terms but not infinite types

That is why infinite [types](#) throw infinite [type error](#).

## Chapter 99

# 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 100

«<Good: Control.Monad.Error ->  
Control.Monad.Except» >

## Chapter 101

# Good: Monad OR Applicative

### 101.0.1 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 -> <$>
>> -> *>
```

### 101.0.2 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`.

### 101.0.3 [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 102

# 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

### 102.1 \*

PVP

Good: PVP

## Chapter 103

# Good: Linear type

Linear types are great to control/minimize resource usage.



## Chapter 104

# 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 105

### Good: Let vs. Where

`let ... in ...` is a separate [expression](#). In contrast, `where` is [bound](#) to a surrounding syntactic [construct](#) (namespace).

## Chapter 106

# Good: RankNTypes

Can powerfully synergyze with [ScopedTypeVariables](#).

## Chapter 107

# Good: Orphan type instance

Practice to address orphan instances:

Does [type class](#) or [type](#) defined by you:

Type class	Type	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 108

# Good: Smart constructor

Only proper smart [constructors](#) should be exported. Do not export [data type constructor](#), only a [type](#).

## Chapter 109

### Good: Thin category

In \* all [morphisms](#) are [epimorphisms](#) and [monomorphisms](#).

## Chapter 110

# 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 111

# Good: Monoid

<>:

[Sets](#) - union.

Maps - left-biased union.

Number - `Sum`, `Product` form separate [monoid categories](#).



## Chapter 112

# 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 113

Good: Use mostly where clauses

## Chapter 114

Good: Where clause is in a scope  
with function parameters

## Chapter 115

Good: Strong preference towards pattern matching over {head, tail, etc.} functions

head and tail and alike [functions](#) are often partial ([unsafe](#)) functions.

## Chapter 116

# 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 117

# 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 118

# 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 119

# 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 120

# 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 121

### 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 122

# 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 123

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 124

# 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 125

# 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 126

# Bad pragma

### 126.1 Bad: Dangerous **LANGUAGE pragma** option

- [DatatypeContexts](#)
- [OverlappingInstances](#)
- [IncoherentInstances](#)
- [ImpredicativeTypes](#)
- [AllowAmbiguousTypes](#)
- [UndecidableInstances](#) - often



## Part VII

Useful **functions** to remember

# Chapter 127

## 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
```

### 127.1 Ord

compare

### 127.2 Calc

div - always makes rounding down, to infinity

divMod - returns a tuple containing the result of integral division and modulo

### 127.3 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 128

### Data.List

`intersperse :: a -> [a] -> [a]` - gets the value and incerts it between values `in` array  
`nub` - remove duplicates from the list

## Chapter 129

# Data.Char

```
ord (Char -> Int)  
chr (Int -> Char)  
isUpper (Char -> Bool)  
toUpper (Char -> Char)
```

## Chapter 130

# QuickCheck

```
quickCheck :: Testable prop => prop -> IO ()
```

```
quickCheck . verbose - run verbose mode
```

## Part VIII

# Tools

## Chapter 131

# ghc-pkg

[List](#) installed packages:

```
ghc-pkg list
```

## Chapter 132

# Search over the Haskell packages code: Codesearch from Aelve

<https://codesearch.aelve.com/>



## Chapter 133

# Integration of NixOS/Nix with Haskell IDE Engine (HIE) and Emacs (Spacemacs)

### 133.1 1. Install the Cachix

Upstream doc: <https://github.com/cachix/cachix>

### 133.2 2. Installation of HIE

Upstream doc: <https://github.com/infinisil/all-hies/#cached-builds>

#### 133.2.1 2.1. Provide cached builds

```
cachix use all-hies
```

#### 133.2.2 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
```

### 133.2.3 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.

## 133.3 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:

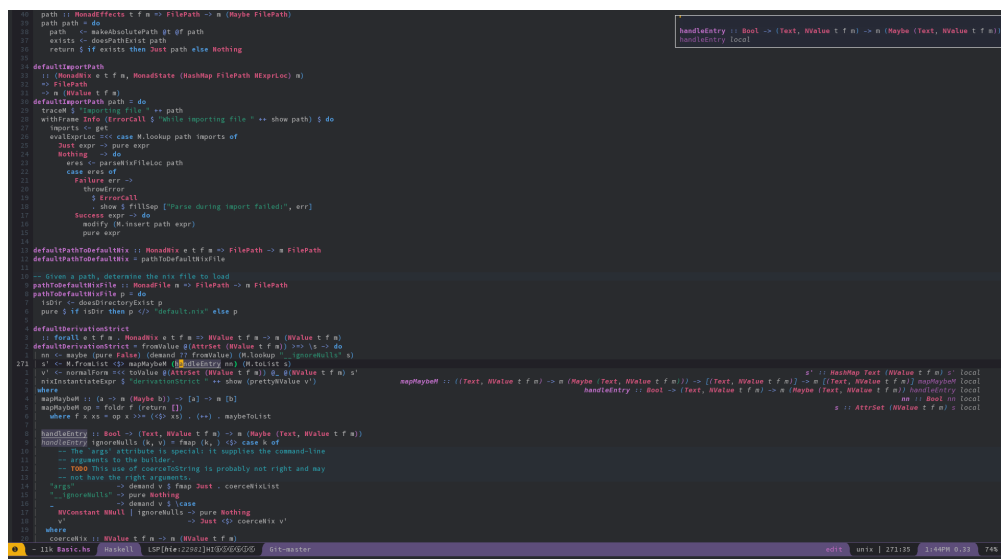
auto-completion configures YASnippet.

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.

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)/

### 133.4 4. Open the Haskell file from a project

133.5 5. Be pleased writing code



## 133.6 6. (optional) Debugging

- ```
readCreateProcess : cabal-helper-wrapper failure
```

282

- a.* If HIE keeps getting ready, failing & restarting - check that the projects `ghc --version` is declared in your `all-hie` NixOS configuration.

## Chapter 134

# 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 135

# 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
```

## Chapter 136

# Continuous integration platrorms (CIs) for Open Source Haskell projets

Since Open Source projects mostly use free tiers of CIs, and different CIs have different features - there is a [constant](#) flux of how to [construct](#) the best possible integration pipeline for Haskell projects.

The current state of affairs is best put in this quote:

Probably the biggest [constraint](#) is whether or not CI needs to test Windows or OS X, since build machines for those are harder to come by. We currently use AppVeyor for Windows builds and Travis for OS X builds since they are free. For Linux you can basically use any CI provider, but in this [case](#) I pay for a Linode VM which I use to host all Dhall-related infrastructure (i.e. all of the \*.dhall-lang.org domains), so I reuse that to host Hydra for Nix-related CI so that I can use more parallelism and more efficient caching to test a wider range of GHC versions on a budget.

For [testing](#) OS X and Windows platforms we use [stack](#). The main reason we don't use Nix for [either](#) platform is that Nix only supports building release binaries on Linux (and even then it's still experimental).

So the basic summary I can give is:

For [testing](#) everything other than cross-platform support: Nix + Linux is best in my opinion

... because you get much more control and intelligent build caching, which is usually [where](#) most CI solutions fall short

For cross-platform support: [stack](#) + whatever CI provider provides free builds for that platform

Also, if you ever can pay for your own NixOS VM and you want to reuse the setup I built, you can find the NixOS configuration for dhall-lang.org here:

<https://github.com/dhall-lang/dhall-lang/tree/master/nixops>



## Part IX

# Libs

## Chapter 137

# Exceptions

- 137.1 **Exceptions** - optionally **pure** extensible **exceptions** that are compatible with the mtl
- 137.2 **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.
- 137.3 **Enclosed-exceptions** - capture **exceptions** from the enclosed computation, while reacting to asynchronous **exceptions** aimed at the calling thread.

## Chapter 138

# Memory management

### 138.1 membrain - [type](#)-safe memory units

## Chapter 139

# Parsers - megaparsec

## Chapter 140

### CLIs - `optparse-`[applicative](#)

## Chapter 141

### HTML - Lucid

## Chapter 142

# Web applications - Servant

## Chapter 143

### IO libraries

- 143.1 Conduit - practical, monolythic, guarantees termination return
- 143.2 Pipes + Pipes Parse - modular, more primitive, theoretically driven



## Chapter 144

### JSON - aeson

## Chapter 145

# 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 146

# 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 exactly 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 147

## 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 148

# Monad transformers and their type classes

## Chapter 149

# 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 150

## Hoogle

### 150.1 Search

Text search ([case](#) insensitive):

- `a`
- `map`
- `con map`

[Type](#) search:

- `:: a`
- `:: a -> a`

Text & [type](#):

`=id a -> a=`

### 150.2 [Scope](#)

#### 150.2.1 Default

[Scope](#) is [Haskell Platform](#) (and [Haskell keywords](#)).

All [Hackage](#) packages are available to search with:

#### 150.2.2 [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`

### 150.2.3 Packages (lower **case**):

- `mode +platform`
- `mode +cmdargs` (only)
- `mode +platform +cmdargs`
- `file -base` (Haskell Platform, excluding the "base" package)

## Chapter 151

# ST-Trick monad

ST is like a [lexical scope](https://wiki.haskell.org/lexical_scope), where all the [variables](https://wiki.haskell.org/variables)/state disappear when the [function](https://wiki.haskell.org/function) returns  
[https://wiki.haskell.ohttps://www.schoolofhaskell.com/school/to-infinity-and-beyond/older-but-still-interesting/deamortized-strg/Monad/ST](https://wiki.haskell.org/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>

### 151.1 \*

ST-Trick

## Chapter 152

# Either

Allows to separate and preserve information about happened, ex. [error](#) handling.

### 152.1 \*

Either data type

## Chapter 153

# 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 154

# Inversion

- a.* Is a [permutation where](#) two elements are out of [order](#).
- b.* See [Inverse](#)



## Chapter 155

# Inverse function

$$f_{x \rightarrow y} \circ (f_{x \rightarrow y})^{-1} = 1_x$$

\*  $\iff$  function is bijective.

Otherwise - partial inverse

## Chapter 156

# 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 157

# Partial inverse

\* when [function](#) is now [bijective](#). When [bijective](#) see [inverse function](#).

## Chapter 158

# PatternSynonyms

Enables [pattern synonym declaration](#), which always begins with the `pattern` word.  
Allows to [abstract](#)-away the [structures](#) of pattern matching.

### 158.1 \*

Pattern synonym  
Pattern synonyms

## Chapter 159

# GHC debug keys

### 159.1 -ddump-ds

Dump desugarer output.

#### 159.1.1 \*

Desugar  
GHC desugar

## Chapter 160

# GHC optimize keys

### 160.1 -foptimal-applicative-do

$O(n^3)$

Always finds optimal [reduction](#) into `<*>` for [ApplicativeDo](#) do notation.

# Chapter 161

## 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 161.1: [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- <a href="#">tensor</a> adjunction               | beta reduction       |
| introduction rule for implication | unit for hom- <a href="#">tensor</a> adjunction                 | eta conversion       |
| logical conjunction               | product                                                         | product type         |
| disjunction                       | coproduct ((-1)-truncation of)                                  | sum type (bifunctor) |
| 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 inductive type |
| 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  |

**161.1** \*

Trinitarism



## Chapter 162

# Techniques functional programming deals with the state

### 162.1 Minimizing

Do not rely on state, try not to change the state. Use it only when it is very necessary.

### 162.2 Concentrating

Concentrate the state in one place.

### 162.3 Deferring

Defer state to the last step of the program, or to external system.

## Chapter 163

# Monadic Error handling

```
(>>=) :: m a -> (a -> m b) -> m b --  $\lambda A.E \vee A$  - computes and drops if error value happens.  
catch :: c a -> (e -> c a) -> c a --  $\lambda E.E \vee A$  - handles "errors" as "normal" values and stops wh
```

## Chapter 164

# 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 165

## 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.

### 165.1 \*

Nothing, Haskell [expressions](#) can't return [Void](#).

Also see: [Maybe](#).

## Chapter 166

# Constructive proof

Method of proof that demonstrates the existence of a mathematical [object](#) by creating or providing a method for creating the [object](#).

## Chapter 167

# 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.

### 167.1 \*

Constructive logic

## Chapter 168

# Principle of explosion

From asserted [statement](#) that contains contradiction - anything can be proven.  
Ancient principle of logic. Both in classical & intuitionistic logic.

### 168.1 \*

Ex falso quodlibet

Ex falso sequitur quodlibet

EFG

Ex contradictione quodlibet

Ex contradictione sequitur quodlibet

ECQ

Deductive explosion

Pseudo-Scotus

## Chapter 169

# 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 170

# 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 171

# Monoidal category, functoriality of ADTs, Profunctors

Category equipped with tensor product.

<>

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 172

# 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:

$\Delta$

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 173

# Arrow in Haskell

```
(->) a b = a -> b
```

`Functorial` in the last `argument` & called Reader `functor`.

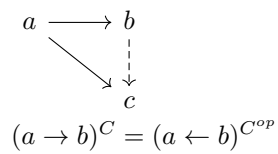
```
newtype Reader c a = Reader (c -> a)
```

```
fmap = ( . )
```

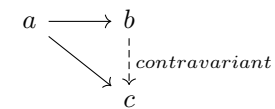
## Chapter 174

# Contravariant functor

```
fmap :: (a -> b) -> Op c a -> Op c b
      (a -> c) -> (b -> c)
```



```
class Contravariant f
  where
    contramap :: (b -> a) -> (f a -> f b)
```



If [arrows](#) does not commute Contravariant functor anyway allows to [construct](#) transformation between these such [arrows](#) to other [arrow](#).

## Chapter 175

# Profunctor

`(->) 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 176

# 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 = (<|>)`

### 176.1 \*

Applicative-Monad proposal  
AMP



# Chapter 177

## Haskell-98

### 177.1 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**,  $\langle \text{tvs} \rangle_{\text{left}} \rightarrow \langle \text{tvs} \rangle_{\text{right}}$ , of the class, every **type variable** in  $S(\langle \text{tvs} \rangle_{\text{right}})$  must appear in  $S(\langle \text{tvs} \rangle_{\text{left}})$ , where  $S$  is the substitution mapping each **type variable** in the class **declaration** to the corresponding **type** in the instance head.

## Chapter 178

# Performance results and comparisons of **types** & solutions

[Haskell performance](#)

## Part XII

# Liturgy

---

*λειτ* <- *λα΄ς* *Laos* the people  
*ουργ΄ς* <- *ργο* *ergon* work  
*λειτουργ΄α* *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.