

# Computational Monotone Co-Design

2025-08-31

# Contents

<b>A. Invitation to computational co-design</b>	<b>13</b>
<b>1. A tour of MCDPL</b>	<b>14</b>
1.1. What MCDPL is . . . . .	14
1.2. Graphical representation . . . . .	14
1.3. Your first model . . . . .	14
1.3.1. Constraining functionality and requirements . . . . .	15
1.3.2. Use of Unicode glyphs in the language . . . . .	16
1.3.3. Aside: a helpful compiler . . . . .	16
1.4. Describing relations between functionality and resources . . . . .	16
1.4.1. Units . . . . .	17
1.5. Catalogs . . . . .	17
1.5.1. Multiple minimal solutions . . . . .	18
1.6. Union/choice of design problems . . . . .	19
1.7. Composing design problems . . . . .	20
1.7.1. Implicit composition of design problems from formulas . . . . .	20
1.7.2. Explicit composition of design problems . . . . .	20
1.7.3. Adding co-design constraints . . . . .	21
1.8. Re-usable design patterns using templates . . . . .	22
<b>B. MCDPL reference</b>	<b>24</b>
<b>2. Introduction</b>	<b>25</b>
2.1. Introduction . . . . .	25
2.1.1. Kinds . . . . .	25
<b>3. Posets and values</b>	<b>26</b>
3.1. <code>poset</code> : Defining finite posets . . . . .	26
3.2. Extrema of posets . . . . .	26
3.2.1. <code>Top</code> and <code>Bottom</code> . . . . .	26
3.2.2. <code>Minimals</code> and <code>Maximals</code> . . . . .	26
3.3. Numerical posets . . . . .	27
3.3.1. Numerical precision . . . . .	27
3.4. Numbers with units . . . . .	27
3.5. Poset products . . . . .	28
3.5.1. <code>x</code> : Anonymous Poset Products . . . . .	28
3.5.2. <code>product</code> : Named Poset Products . . . . .	28
3.6. Posets of subsets . . . . .	29
3.6.1. <code>powerset</code> : Power sets . . . . .	29
3.6.2. <code>EmptySet</code> : The empty set . . . . .	29
3.6.3. <code>UpperSets</code> and <code>LowerSets</code> : Upper and lower sets . . . . .	29
3.7. Defining uncertain constants . . . . .	30
<b>4. Named DPs</b>	<b>31</b>
4.1. Defining NDPs . . . . .	31
4.2. Constructing NDPs as catalogs . . . . .	31
4.2.1. True and false . . . . .	32
4.3. Constructing NDPs from YAML files . . . . .	32
4.4. Describing Monotone Co-Design Problems . . . . .	33
4.4.1. Declaring functionality and requirements explicitly . . . . .	33
4.4.2. Declaring functionality and requirements implicitly using expressions . . . . .	34
4.4.3. Forwarding functionalities and requirements from other subproblems . . . . .	34
4.4.4. Declaring functionality and requirements using interfaces . . . . .	34
4.4.5. Ignoring or propagating the functionality/requirements of subproblems . . . . .	35

4.4.6.	Shortcuts for summing over functionalities and requirements	36
4.5.	Mathematical relations between functionalities and requirements	37
4.5.1.	Abstract interpretation of mathematical relations	37
4.5.2.	Min and max	39
4.5.3.	Floor and ceil relations	39
4.5.4.	Built-in approximations	40
4.6.	Accessing the components of a product	40
4.6.1.	Accessing the components of an anonymous product	40
4.7.	Operations on NDPS	41
4.7.1.	<code>compact</code> : Compactification	41
4.7.2.	<code>flatten</code> : Flattening	42
4.7.3.	<code>abstract</code> : Abstraction	43
4.7.4.	<code>canonical</code> : Canonical form	43
4.8.	<code>choose</code> : Union of design problems	43
<b>5.</b>	<b>Higher-order modeling</b>	<b>44</b>
5.1.	Interfaces	44
5.1.1.	<code>interface</code> : Declaring interfaces	44
5.1.2.	<code>extends</code> : Extending interfaces	44
5.1.3.	<code>implements</code> : Using interfaces when defining models	44
5.2.	Templates	45
5.2.1.	<code>template</code> : Declaring templates	45
5.2.2.	<code>specialize</code> : Instantiating templates	45
<b>6.</b>	<b>Queries</b>	<b>46</b>
6.1.	Defining queries	46
6.1.1.	<code>FixFunMinRes</code>	46
6.1.2.	<code>FixResMaxFun</code>	47
<b>7.</b>	<b>Syntax</b>	<b>48</b>
7.1.	MCDPL syntax	48
7.1.1.	Characters	48
7.1.2.	Comments	48
7.1.3.	Reserved keywords	48
7.1.4.	Syntactic equivalence	48
7.1.5.	Identifiers	48
7.1.6.	Use of Greek letters as part of identifiers	49
<b>C.</b>	<b>Software manual</b>	<b>50</b>
<b>8.</b>	<b>MCDP Command line interface</b>	<b>51</b>
8.1.	Installation	51
8.1.1.	Prerequisites	51
8.1.2.	🐧 Linux (Ubuntu/Debian)	51
8.1.3.	🍏 macOS	51
8.1.4.	🪟 Windows (Experimental)	52
8.1.5.	Getting Started	53
8.1.6.	Troubleshooting	53
8.2.	<code>mcdp update</code>	53
8.3.	<code>mcdp co-design plot</code>	53
8.4.	<code>mcdp co-design solve</code>	54
8.4.1.	Resolution options	55
8.4.2.	Implementation options	55
8.5.	<code>mcdp co-design solve-query</code>	55
8.5.1.	Resolution options	55
8.5.2.	<code>--imp</code> - Computing implementations	55
8.5.3.	<code>--blueprints</code> - Computing blueprints	55
8.6.	<code>mcdp co-design export</code>	56
<b>9.</b>	<b>Libraries for parsing the exported data</b>	<b>57</b>
9.1.	🐍 Python library <code>mcdp-format2-py</code>	57
9.2.	🦀 Rust crate <code>mcdp-format2-rs</code>	57

<b>D. Mathematical underpinnings for computational co-design</b>	<b>58</b>
<b>10. Order theory</b>	<b>59</b>
10.1. Posets	59
10.2. Special subsets of posets	59
10.3. Monotone maps	59
10.4. Closure operators	59
10.4.1. Upper and lower closure of a point	59
10.4.2. Upper and lower closure of a subset	60
10.4.3. Upper and lower closure of a function	60
<b>11. Design problems (DPs)</b>	<b>62</b>
11.1. Design problems	62
11.2. <b>Pos<sub>L</sub></b> and <b>Pos<sub>U</sub></b>	62
11.3. Queries for DPs	62
<b>12. DP computability and well foundedness</b>	<b>64</b>
12.1. Well-foundedness	64
12.1.1. Well-foundedness of upper and lower sets	64
12.1.2. Well-foundedness of <b>Pos<sub>U</sub></b> and <b>Pos<sub>L</sub></b> morphisms	64
12.1.3. Well-foundedness of DPs	65
12.2. Lifting maps to DPs	65
12.3. Upper and lower preimage of a monotone map	66
12.4. Well-foundedness and Galois connections	67
12.5. Well-foundedness and Scott-continuity	68
12.5.1. Scott-continuity	68
12.5.2. Co-Scott-continuity	68
12.5.3. Scott-continuity and well-foundedness	69
12.6. Lifting as functors	70
12.6.1. Restriction to Scott-(co)continuous maps	71
<b>13. Scalable computation for DPs</b>	<b>72</b>
13.1. Scalable maps	72
13.2. Approximation of DP queries	72
<b>14. Design problems with implementations (DPIs)</b>	<b>75</b>
14.1. DPIs	75
14.2. Optimization queries associated to a DPI	76
14.3. Categories <b>Pos<sub>UI</sub></b> and <b>Pos<sub>LI</sub></b>	77
14.3.1. Relating <b>Pos<sub>UI</sub></b> and <b>Pos<sub>LI</sub></b> to <b>Pos<sub>U</sub></b> and <b>Pos<sub>L</sub></b>	78
14.3.2. Pre-order on <b>Pos<sub>LI</sub></b> and <b>Pos<sub>UI</sub></b>	78
14.4. DPI queries as <b>Pos<sub>UI</sub>/Pos<sub>LI</sub></b> morphisms	79
14.5. Free-forgetful adjunction between DP and DPI	80
<b>15. Scalable computation for DPI</b>	<b>81</b>
15.1. <b>SPos<sub>UI</sub></b> and <b>SPos<sub>LI</sub></b>	81
15.2. Approximation of DPI queries	81
<b>16. Numerical approximation</b>	<b>83</b>
16.1. Approximation of DPs	83
16.2. Approximation of operations in complete lattices	83
16.2.1. Upper and lower approximations of the original operation	83
16.2.2. Comparing the DPs	84
<b>E. Catalogs</b>	<b>85</b>
<b>17. Sets and posets catalog</b>	<b>86</b>
17.1. Sets constructions	86
17.1.1. Cartesian products	86
17.1.2. Sum	86
17.2. Constructions for single posets	86
17.2.1. Opposite of a poset	86



17.2.2.	Arrow constructions . . . . .	86
17.2.3.	Discretized version of a poset . . . . .	87
17.2.4.	Posets of subsets . . . . .	87
17.3.	Constructions with multiple posets . . . . .	87
17.3.1.	Cartesian product of posets . . . . .	87
17.3.2.	Direct sum of posets . . . . .	87
17.3.3.	Lexicographic product of posets . . . . .	88
17.4.	Poset Filters . . . . .	88
17.4.1.	Finite subposet of an ambient poset . . . . .	88
17.4.2.	Interval in a poset . . . . .	88
17.4.3.	Lower and upper closure in a poset . . . . .	88
17.4.4.	Union and Intersection of sub posets . . . . .	89
17.4.5.	Sampling a poset . . . . .	89
<b>18.</b>	<b>Monotone maps catalog</b>	<b>90</b>
18.1.	Identity map . . . . .	90
18.2.	Constant maps . . . . .	90
18.3.	Ceiling and floor . . . . .	90
18.3.1.	Generalized rounding . . . . .	90
18.4.	Sum, multiplication, and division . . . . .	92
18.4.1.	Sum . . . . .	92
18.4.2.	Multiplication . . . . .	93
18.4.3.	Division . . . . .	95
18.5.	Unary join and meet operations . . . . .	97
18.6.	$n$ -ary joins and meets . . . . .	99
18.6.1.	$n$ -ary Join . . . . .	99
18.6.2.	$n$ -ary Meet . . . . .	100
18.7.	Lifts to subsets . . . . .	100
18.8.	Plumbing . . . . .	101
18.8.1.	Slicing . . . . .	101
18.8.2.	Injectons . . . . .	101
18.9.	Catalog . . . . .	102
18.10.	Threshold maps . . . . .	102
18.11.	Tests . . . . .	103
18.11.1.	constant $\leq x$ . . . . .	103
18.11.2.	constant $< x$ . . . . .	104
18.11.3.	$x \leq$ constant . . . . .	104
18.11.4.	$x <$ constant . . . . .	105
18.12.	Lower/upper set containment tests . . . . .	105
18.12.1.	Lower set containment tests . . . . .	105
18.12.2.	Upper set containment tests . . . . .	106
18.13.	Order as a function . . . . .	106
<b>19.</b>	<b>Monotone map compositions catalog</b>	<b>107</b>
19.1.	Constructions for single maps . . . . .	107
19.1.1.	Opposite of a map . . . . .	107
19.2.	Constructions for multiple maps . . . . .	107
19.2.1.	Parallel composition . . . . .	107
19.2.2.	Series composition . . . . .	108
19.2.3.	Product of maps . . . . .	108
19.2.4.	Sum of maps . . . . .	109
19.2.5.	Coproduct of maps . . . . .	109
19.2.6.	Domain union . . . . .	110
<b>20.</b>	<b>LPos and UPos catalog</b>	<b>111</b>
20.1.	Identity morphisms . . . . .	111
20.2.	Lifting maps . . . . .	111
20.3.	Catalog maps . . . . .	111
20.4.	Union and intersection of principal lower sets . . . . .	112
20.5.	Representing principal lower and upper sets . . . . .	113
20.6.	Generic inverses for mathematical operations . . . . .	113
20.7.	Filtering . . . . .	114

20.8.	Parallel composition . . . . .	115
20.9.	Sum . . . . .	115
20.10.	Codomain Sum . . . . .	115
20.11.	Product of maps . . . . .	116
20.12.	Series composition . . . . .	116
20.13.	Union and Intersection of maps . . . . .	117
20.14.	Trace . . . . .	117
<b>21.</b>	<b>LPosl and UPosl catalog</b>	<b>119</b>
21.1.	Identity . . . . .	119
21.2.	Constant maps . . . . .	119
21.3.	Catalog maps . . . . .	119
21.4.	Lifting maps . . . . .	120
21.5.	Series composition . . . . .	121
21.6.	Parallel composition . . . . .	121
21.7.	Intersection of maps . . . . .	122
21.8.	Union of maps . . . . .	122
21.9.	Transforming maps . . . . .	123
21.10.	Trace . . . . .	123
<b>22.</b>	<b>SLPos and SUPos catalog</b>	<b>125</b>
22.1.	Identities . . . . .	125
22.2.	Lifting . . . . .	125
22.3.	Parallel composition . . . . .	126
22.4.	Series composition . . . . .	126
22.5.	Union . . . . .	127
22.6.	Intersection . . . . .	128
22.7.	Trace . . . . .	128
22.8.	Product . . . . .	129
22.9.	Sum . . . . .	129
22.10.	Product intersection . . . . .	131
22.11.	Scalable inverse of sum and multiplication operations . . . . .	131
22.12.	Explicit approximation . . . . .	132
<b>23.</b>	<b>SLPosl and SUPosl catalog</b>	<b>134</b>
23.1.	Lifts . . . . .	134
23.2.	Explicit approximations . . . . .	134
23.3.	Parallel composition . . . . .	135
23.4.	Series composition . . . . .	136
23.5.	Intersection . . . . .	136
23.6.	Union . . . . .	137
23.7.	Trace . . . . .	138
<b>24.</b>	<b>DP catalog</b>	<b>139</b>
24.1.	Identity . . . . .	139
24.2.	Ambient conversion . . . . .	139
24.3.	Isomorphism . . . . .	139
24.4.	Lower lift of a map . . . . .	140
24.5.	Upper lift of a map . . . . .	140
24.6.	Functionalities/requirements limits . . . . .	141
24.6.1.	Functionality not more than the requirement and constant . . . . .	141
24.6.2.	Requirement not less than the functionality and constant . . . . .	141
24.6.3.	All functionalities less than the requirement . . . . .	141
24.6.4.	Any functionality less than the requirement . . . . .	142
24.6.5.	All requirements more than the functionality . . . . .	142
24.6.6.	Any requirement more than the functionality . . . . .	142
24.6.7.	All constants less than the requirement . . . . .	142
24.6.8.	Functionality less than all constants . . . . .	142
24.6.9.	Functionality and all constants less than the requirement . . . . .	143
24.6.10.	Functionality or any constant less than the requirement . . . . .	143
24.6.11.	Functionality less than the requirement and all constants . . . . .	143
24.6.12.	Functionality less than the requirement or any constant . . . . .	143

<b>25. DPI catalog</b>	<b>144</b>
25.1. True and false	144
25.1.1. True	144
25.1.2. False	144
25.2. Catalogs	145
25.3. Parallel composition	146
25.4. Series	147
25.5. Intersection	148
25.6. Union	149
25.7. Trace	150

## F. MCDP Format 2 152

<b>26. Top level</b>	<b>153</b>
26.1. Root - Top-level object types for what can be serialized in a file.	154
26.2. Poset - A poset.	155
26.2.1. P_Bool - The poset of boolean values	155
26.2.2. P_Decimal - Decimal numbers with fixed precision.	156
26.2.3. P_Finite - Arbitrary finite poset	156
26.2.4. P_Float - Poset of floating point numbers.	157
26.2.5. P_Fractions - Fractions with a maximum absolute value for numerator and denominator.	157
26.2.6. P_Integer - Poset of integers.	157
26.2.7. P_Unknown - Placeholder for an unknown poset	158
26.2.8. P_C_Arrow - Arrow constructors for posets.	158
26.2.9. P_C_Discretized - Discretized version of a poset.	158
26.2.10. P_C_LowerSets - The poset of lower sets of a given poset.	159
26.2.11. P_C_Opposite - Opposite of a poset	159
26.2.12. P_C_Power - Power poset of a given poset.	160
26.2.13. P_C_Twisted - Twisted arrow construction of a poset.	160
26.2.14. P_C_Units - A poset with units	160
26.2.15. P_C_UpperSets - The poset of upper sets of a given poset.	161
26.2.16. P_C_Lexicographic - Lexicographic product of posets	161
26.2.17. P_C_Product - Cartesian product of posets	162
26.2.18. P_C_ProductSmash - Poset smash product	162
26.2.19. P_C_Sum - Direct sum of posets.	163
26.2.20. P_C_SumSmash - Direct (smash) sum of posets	163
26.2.21. P_F_Bounded - A subposet that allows to sample a numeric poset.	164
26.2.22. P_F_C_Intersection - Intersection of posets.	165
26.2.23. P_F_C_Union - Union of posets	165
26.2.24. P_F_Interval - An interval in a poset.	166
26.2.25. P_F_LowerClosure - Lower closure in a poset.	166
26.2.26. P_F_Subposet - A finite subposet of an ambient poset.	166
26.2.27. P_F_UpperClosure - Upper closure in a poset.	167
26.3. MonotoneMap - Monotone maps	168
26.3.1. M_Constant - A constant function	169
26.3.2. M_Empty - The unique map from the empty set to another	169
26.3.3. M_Explicit - A map defined pointwise.	169
26.3.4. M_Id - Identity map	169
26.3.5. M_Undefined - Undefined map	170
26.3.6. M_Unknown - Placeholder for an unknown map	170
26.3.7. M_ContainedInLowerSet - Test for containment in a lower set	170
26.3.8. M_ContainedInUpperSet - Test for containment in an upper set	170
26.3.9. M_Injection - Injection into a poset sum	170
26.3.10. M_Join - Join operation	171
26.3.11. M_JoinConstant - Join with a constant value	171
26.3.12. M_Meet - Meet operation	171
26.3.13. M_MeetConstant - Meet with a constant	172
26.3.14. M_RepresentPrincipalLowerSet_TotalOrderBounded - Largest principal lower set in the poset.	172
26.3.15. M_RepresentPrincipalUpperSet_TotalOrderBounded - Largest principal upper set in the poset.	172
26.3.16. M_SmashInjection - Injection into a smash sum	172
26.3.17. M_C_Op - Opposite of a map	173

26.3.18.	<a href="#">M_C_RefineDomain</a> - A refinement of the domain of a monotone map	173
26.3.19.	<a href="#">M_C_WrapUnits</a> - Wraps a monotone map with units descriptions for domain and codomain.	173
26.3.20.	<a href="#">M_C_Coproduct</a> - Coproduct of monotone maps	173
26.3.21.	<a href="#">M_C_CoproductSmash</a> - Smash coproduct of two monotone maps	174
26.3.22.	<a href="#">M_C_DomProdCodSmash</a> - A monotone map from a product of domains to a smash product of codomains.	174
26.3.23.	<a href="#">M_C_DomSmashCodProd</a> - A monotone map from the smash product of domains to the product of codomains.	174
26.3.24.	<a href="#">M_C_DomUnion</a> - Domain union of monotone maps	175
26.3.25.	<a href="#">M_C_Parallel</a> - Monoidal product of monotone maps	175
26.3.26.	<a href="#">M_C_ParallelSmash</a> - Monoidal (smash) product of monotone maps	175
26.3.27.	<a href="#">M_C_Product</a> - Product of monotone maps	175
26.3.28.	<a href="#">M_C_ProductSmash</a> - Smash product of monotone maps	176
26.3.29.	<a href="#">M_C_Series</a> - Series composition of monotone maps	176
26.3.30.	<a href="#">M_C_Sum</a> - Sum of monotone maps	176
26.3.31.	<a href="#">M_C_SumSmash</a> - Smash sum of monotone maps	177
26.3.32.	<a href="#">M_AddL</a> - Addition in the L topology.	177
26.3.33.	<a href="#">M_AddLConstant</a> - Add a constant in the L topology.	177
26.3.34.	<a href="#">M_AddU</a> - Addition in the U topology.	177
26.3.35.	<a href="#">M_AddUConstant</a> - Addition of constant in the U topology.	178
26.3.36.	<a href="#">M_Ceil0</a> - Ceiling function relative	178
26.3.37.	<a href="#">M_DivideLConstant</a> - Division by a constant (L topology)	178
26.3.38.	<a href="#">M_DivideUConstant</a> - Division by a constant (U topology)	178
26.3.39.	<a href="#">M_Floor0</a> - Floor function relative	179
26.3.40.	<a href="#">M_MultiplyL</a> - Multiplication (L topology)	179
26.3.41.	<a href="#">M_MultiplyLConstant</a> - Multiplication by a constant (L topology)	179
26.3.42.	<a href="#">M_MultiplyU</a> - Multiplication (U topology)	179
26.3.43.	<a href="#">M_MultiplyUConstant</a> - Multiplication by a constant (U topology)	179
26.3.44.	<a href="#">M_PowerFracL</a> - Lift to the power of a fraction (L topology)	180
26.3.45.	<a href="#">M_PowerFracU</a> - Lift to the power of a fraction (U topology)	180
26.3.46.	<a href="#">M_RoundDown</a> - Round down	180
26.3.47.	<a href="#">M_RoundUp</a> - Round up	180
26.3.48.	<a href="#">M_ScaleL</a> - Scaling in the L topology by a fraction.	180
26.3.49.	<a href="#">M_ScaleU</a> - Scaling in the U topology by a fraction.	181
26.3.50.	<a href="#">M_SubLConstant</a> - Subtraction of a constant (L topology)	181
26.3.51.	<a href="#">M_SubUConstant</a> - Subtraction by a constant (U topology)	181
26.3.52.	<a href="#">M_C_LiftToSubsets</a> - Lift of a monotone map to subsets	181
26.3.53.	<a href="#">M_LiftToLowerSets</a> - Lifts a monotone map to lower sets	181
26.3.54.	<a href="#">M_LiftToUpperSets</a> - Lifts a monotone map to upper sets	182
26.3.55.	<a href="#">M_BottomIfNotTop</a> - Maps top to top, and everything else to bottom.	182
26.3.56.	<a href="#">M_IdentityBelowThreshold</a> - A monotone map that outputs a constant value if the input is above a threshold.	182
26.3.57.	<a href="#">M_Threshold1</a> - Threshold map (r-to-f for DP_FuncNotMoreThan)	182
26.3.58.	<a href="#">M_Threshold2</a> - Threshold map (f-to-r for DP_ResNotLessThan)	183
26.3.59.	<a href="#">M_TopIfNotBottom</a> - Maps bottom to bottom, and everything else to top.	183
26.3.60.	<a href="#">M_Lift</a> - Lifts a value to a tuple with one element.	183
26.3.61.	<a href="#">M_TakeIndex</a> - Projection of an element in a poset product.	183
26.3.62.	<a href="#">M_TakeRange</a> - Projection of a range of elements in a smash poset product.	184
26.3.63.	<a href="#">M_Unlift</a> - Unlifts a one-element tuple to its single element.	184
26.3.64.	<a href="#">M_C_Leq_X</a> - Tests constant $\leq x$	184
26.3.65.	<a href="#">M_C_Lt_X</a> - Tests constant $< x$	184
26.3.66.	<a href="#">M_X_Leq_C</a> - Tests $x \leq$ constant	185
26.3.67.	<a href="#">M_X_Lt_C</a> - Tests $x <$ constant	185
26.3.68.	<a href="#">M_Leq</a> - Tests $x_1 \leq_p x_2$	185
26.4.	<a href="#">L1Map</a> - Map to lower sets of functionalities.	186
26.4.1.	<a href="#">L1_Constant</a> - Constant map	186
26.4.2.	<a href="#">L1_Entire</a> - Returns the entire poset	187
26.4.3.	<a href="#">L1_Explicit</a> - Map defined pointwise	187
26.4.4.	<a href="#">L1_Identity</a> - Lift of the identity map	187
26.4.5.	<a href="#">L1_Unknown</a> - Placeholder for an unknown map.	187
26.4.6.	<a href="#">L1_Catalog</a> - Map induced by a catalog of options.	188
26.4.7.	<a href="#">L1_IntersectionOfPrinLowerSets</a> - Intersection of principal lower sets.	188
26.4.8.	<a href="#">L1_RepresentPrincipallLowerSet</a> - Represent a principal lower set	188
26.4.9.	<a href="#">L1_UnionOfPrinLowerSets</a> - Union of principal lower sets.	188
26.4.10.	<a href="#">L1_C_CodSum</a> - Co-domain sum combination	189

26.4.11.	<a href="#">L1_C_CodSumSmash</a> - Co-domain (smash) sum combination	189
26.4.12.	<a href="#">L1_C_DomUnion</a> - Domain union	189
26.4.13.	<a href="#">L1_C_Parallel</a> - Monoidal product	190
26.4.14.	<a href="#">L1_C_ProdIntersection</a> - From product to intersection	190
26.4.15.	<a href="#">L1_C_Product</a> - Product	190
26.4.16.	<a href="#">L1_C_Series</a> - Series composition	190
26.4.17.	<a href="#">L1_C_Intersection</a> - Intersection	191
26.4.18.	<a href="#">L1_C_Union</a> - Union	191
26.4.19.	<a href="#">L1_C_RefineDomain</a> - Refines the domain of a monotone map.	191
26.4.20.	<a href="#">L1_C_Trace</a> - Trace	191
26.4.21.	<a href="#">L1_C_WrapUnits</a> - Decorates a map with units.	192
26.4.22.	<a href="#">L1_InvMul_Opt</a> - Finite-resolution optimistic approximation of the inverse of a multiplication map.	192
26.4.23.	<a href="#">L1_InvMul_Pes</a> - Finite-resolution pessimistic approximation of the inverse of an addition map.	192
26.4.24.	<a href="#">L1_InvSum_Opt</a> - Finite-resolution optimistic approximation of the inverse of a multiplication map.	192
26.4.25.	<a href="#">L1_InvSum_Pes</a> - Finite-resolution pessimistic approximation of the inverse of an addition map.	193
26.4.26.	<a href="#">L1_FromFilter</a> - Filters based on a monotone map.	193
26.4.27.	<a href="#">L1_L_Linv</a> - Lower inverse of a monotone map	193
26.4.28.	<a href="#">L1_Lift</a> - Lifts a monotone map	193
26.4.29.	<a href="#">L1_TopAlternating</a> - Lower inverse for the meet map	194
26.5.	<a href="#">U1Map</a> - Map to upper sets of resources.	195
26.5.1.	<a href="#">U1_Constant</a> - Constant map	195
26.5.2.	<a href="#">U1_Entire</a> - Returns the entire poset	196
26.5.3.	<a href="#">U1_Explicit</a> - Map defined pointwise	196
26.5.4.	<a href="#">U1_Identity</a> - Lift of the identity map	196
26.5.5.	<a href="#">U1_Unknown</a> - Placeholder for an unknown map.	196
26.5.6.	<a href="#">U1_Catalog</a> - Map induced by a catalog of options.	197
26.5.7.	<a href="#">U1_IntersectionOfPrinUpperSets</a> - Intersection of principal upper sets.	197
26.5.8.	<a href="#">U1_RepresentPrincipalUpperSet</a> - Represent a principal upper set	198
26.5.9.	<a href="#">U1_UnionOfPrinUpperSets</a> - Union of principal upper sets.	198
26.5.10.	<a href="#">U1_C_CodSum</a> - Co-domain sum combination	198
26.5.11.	<a href="#">U1_C_CodSumSmash</a> - Co-domain (smash) sum combination	199
26.5.12.	<a href="#">U1_C_DomUnion</a> - Domain union	199
26.5.13.	<a href="#">U1_C_Parallel</a> - Monoidal product	199
26.5.14.	<a href="#">U1_C_ProdIntersection</a> - From product to intersection	200
26.5.15.	<a href="#">U1_C_Product</a> - Product	200
26.5.16.	<a href="#">U1_C_Series</a> - Series composition	200
26.5.17.	<a href="#">U1_C_Intersection</a> - Intersection	200
26.5.18.	<a href="#">U1_C_Union</a> - Union	201
26.5.19.	<a href="#">U1_C_RefineDomain</a> - Refines the domain of a monotone map.	201
26.5.20.	<a href="#">U1_C_Trace</a> - Trace	201
26.5.21.	<a href="#">U1_C_WrapUnits</a> - Decorates a map with units.	201
26.5.22.	<a href="#">U1_InvMul_Opt</a> - Finite-resolution optimistic approximation of the inverse of a multiplication map.	202
26.5.23.	<a href="#">U1_InvMul_Pes</a> - Finite-resolution pessimistic approximation of the inverse of a multiplication map.	202
26.5.24.	<a href="#">U1_InvSum_Opt</a> - Finite-resolution optimistic approximation of the inverse of an addition map.	202
26.5.25.	<a href="#">U1_InvSum_Pes</a> - Finite-resolution pessimistic approximation of the inverse of an addition map.	202
26.5.26.	<a href="#">U1_FromFilter</a> - Filters based on a monotone map.	203
26.5.27.	<a href="#">U1_L_Uinv</a> - Computes the upper inverse of a monotone map.	203
26.5.28.	<a href="#">U1_Lift</a> - Lifts a monotone map	203
26.5.29.	<a href="#">U1_Uinv_Join</a>	203
26.5.30.	<a href="#">U1_Uinv_JoinConstant</a>	204
26.6.	<a href="#">LMap</a> - Map to lower sets of functionalities and implementations.	205
26.6.1.	<a href="#">L_Constant</a> - Constant map	205
26.6.2.	<a href="#">L_Identity</a> - Identity morphism	205
26.6.3.	<a href="#">L_Unknown</a> - Placeholder for an unknown map	205
26.6.4.	<a href="#">L_Catalog</a> - LMap for a catalog	206
26.6.5.	<a href="#">L_C_Parallel</a> - Monoidal product	206
26.6.6.	<a href="#">L_C_Series</a> - Series composition	206
26.6.7.	<a href="#">L_C_Intersection</a> - Intersection of maps	206
26.6.8.	<a href="#">L_C_Union</a> - Union of maps	207
26.6.9.	<a href="#">L_C_ITransform</a> - Transforms the implementation of another map.	207
26.6.10.	<a href="#">L_C_RefineDomain</a> - Refines the domain of a monotone map	207
26.6.11.	<a href="#">L_C_Trace</a> - Trace	207

26.6.12.	<code>L_C_WrapUnits</code> - Decorates a map with units. . . . .	207
26.6.13.	<code>L_L_Lift1_Constant</code> - Lifts a L1Map morphisms with a constant value for the implementation. . . . .	208
26.6.14.	<code>L_L_Lift1_Transform</code> - Lifts a L1Map morphism with a function to compute the implementation. . . . .	208
26.7.	<code>UMap</code> - Map to upper sets of resources and implementations. . . . .	209
26.7.1.	<code>U_Constant</code> - Constant map . . . . .	209
26.7.2.	<code>U_Identity</code> - Identity . . . . .	209
26.7.3.	<code>U_Unknown</code> - Placeholder for an unknown map . . . . .	209
26.7.4.	<code>U_Catalog</code> - UMap for a catalog . . . . .	210
26.7.5.	<code>U_C_Parallel</code> - Monoidal product . . . . .	210
26.7.6.	<code>U_C_Series</code> - Series composition . . . . .	210
26.7.7.	<code>U_C_Intersection</code> - Intersection of maps . . . . .	210
26.7.8.	<code>U_C_Union</code> - Union of maps . . . . .	211
26.7.9.	<code>U_C_ITransform</code> - Transforms the implementation of another map. . . . .	211
26.7.10.	<code>U_C_RefineDomain</code> - Refines the domain of a monotone map . . . . .	211
26.7.11.	<code>U_C_Trace</code> - Trace . . . . .	211
26.7.12.	<code>U_C_WrapUnits</code> - Decorates a map with units. . . . .	211
26.7.13.	<code>U_L_Lift1_Constant</code> - Lifts a U1Map morphism with a constant value for the implementation. . . . .	212
26.7.14.	<code>U_L_Lift1_Transform</code> - Lifts a U1Map morphism with a function to compute the implementation. . . . .	212
26.8.	<code>SL1Map</code> - Scalable map to lower sets of functionalities. . . . .	213
26.8.1.	<code>SL1_C_Parallel</code> - Monoidal product . . . . .	213
26.8.2.	<code>SL1_C_Series</code> - Series composition . . . . .	213
26.8.3.	<code>SL1_Identity</code> - Identity . . . . .	214
26.8.4.	<code>SL1_Unknown</code> - Placeholder for an unknown SL1Map . . . . .	214
26.8.5.	<code>SL1_C_CodSum</code> - Sum of maps . . . . .	214
26.8.6.	<code>SL1_C_CodSumSmash</code> - Smash sum . . . . .	214
26.8.7.	<code>SL1_C_ProdIntersection</code> - Product of domains, intersection of codomains . . . . .	214
26.8.8.	<code>SL1_C_Product</code> - Product of SL1 maps . . . . .	215
26.8.9.	<code>SL1_C_Intersection</code> - Intersection of SL1 maps . . . . .	215
26.8.10.	<code>SL1_C_Union</code> - Union of SL1 maps . . . . .	215
26.8.11.	<code>SL1_C_RefineDomain</code> - Refinement of the domain . . . . .	215
26.8.12.	<code>SL1_C_Trace</code> - Trace . . . . .	216
26.8.13.	<code>SL1_C_WrapUnits</code> - Decorates a map with units for the domain and codomain. . . . .	216
26.8.14.	<code>SL1_Exact</code> - Lifts a L1Map to a SL1Map. . . . .	216
26.8.15.	<code>SL1_InvMultiply</code> - The lower inverse of multiplication. . . . .	216
26.8.16.	<code>SL1_InvSum</code> - The lower inverse of addition. . . . .	216
26.8.17.	<code>SL1_C_ExplicitApprox</code> - Constructs a SL1Map from explicit approximations of L1Map maps. . . . .	217
26.9.	<code>SU1Map</code> - Scalable map to upper sets of resources. . . . .	218
26.9.1.	<code>SU1_C_Parallel</code> - Monoidal product . . . . .	218
26.9.2.	<code>SU1_C_Series</code> - Series composition . . . . .	218
26.9.3.	<code>SU1_Identity</code> - Identity . . . . .	219
26.9.4.	<code>SU1_Unknown</code> - Placeholder for an unknown SU1Map . . . . .	219
26.9.5.	<code>SU1_C_CodSum</code> - Sum of maps . . . . .	219
26.9.6.	<code>SU1_C_CodSumSmash</code> - Smash sum . . . . .	219
26.9.7.	<code>SU1_C_ProdIntersection</code> - Product of domains, intersection of codomains . . . . .	219
26.9.8.	<code>SU1_C_Product</code> - Product of SU1 maps . . . . .	220
26.9.9.	<code>SU1_C_Intersection</code> - Intersection of SU1 maps . . . . .	220
26.9.10.	<code>SU1_C_Union</code> - Union of SU1 maps . . . . .	220
26.9.11.	<code>SU1_C_RefineDomain</code> - Refinement of the domain . . . . .	220
26.9.12.	<code>SU1_C_Trace</code> - Trace . . . . .	221
26.9.13.	<code>SU1_C_WrapUnits</code> - Wraps a map with units. . . . .	221
26.9.14.	<code>SU1_Exact</code> - Lifts a U1Map to a SU1Map. . . . .	221
26.9.15.	<code>SU1_InvMultiply</code> - The upper inverse of multiplication. . . . .	221
26.9.16.	<code>SU1_InvSum</code> - The inverse of addition. . . . .	221
26.9.17.	<code>SU1_C_ExplicitApprox</code> - Constructs a SU1Map from explicit approximations of U1Map maps. . . . .	222
26.10.	<code>SLMap</code> - Scalable map to lower sets of functionalities and implementations. . . . .	223
26.10.1.	<code>SL_Identity</code> - Identity . . . . .	223
26.10.2.	<code>SL_Unknown</code> - Placeholder for unknown SLMap . . . . .	223
26.10.3.	<code>SL_C_Intersection</code> - Intersection of the results of a set of maps. . . . .	223
26.10.4.	<code>SL_C_Parallel</code> - Monoidal product . . . . .	224
26.10.5.	<code>SL_C_Series</code> - Series composition . . . . .	224
26.10.6.	<code>SL_C_Union</code> - Composition of SLMaps using the union of the results. . . . .	224
26.10.7.	<code>SL_C_ITransform</code> - Transforms the implementations of a SLMap. . . . .	224



26.10.8.	<a href="#">SL_C_RefineDomain</a>	- Refines the domain of another SLMaP	225
26.10.9.	<a href="#">SL_C_Trace</a>	- Trace of a SLMaP.	225
26.10.10.	<a href="#">SL_C_WrapUnits</a>	- Decorates with units another SLMaP.	225
26.10.11.	<a href="#">SL_L_Exact</a>	- Lifts a LMaP to a SLMaP.	225
26.10.12.	<a href="#">SL_L_Explicit_Approx</a>	- Construct a SLMaP from explicit optimistic and pessimistic approximations.	225
26.10.13.	<a href="#">SL_L_Lift1_Constant</a>	- Lifts a SL1MaP to SLMaP with a constant implementation.	226
26.10.14.	<a href="#">SL_L_Lift1_Transform</a>	- Lifts a SL1MaP to SLMaP by generating the implementations.	226
26.11.	<a href="#">SUMaP</a>	- Scalable map to upper sets of resources and implementations.	227
26.11.1.	<a href="#">SU_Identity</a>	- Identity	227
26.11.2.	<a href="#">SU_Unknown</a>	- Placeholder for unknown SUMaP	227
26.11.3.	<a href="#">SU_C_Intersection</a>	- Intersection of the results of a set of maps.	227
26.11.4.	<a href="#">SU_C_Parallel</a>	- Monoidal product	228
26.11.5.	<a href="#">SU_C_Series</a>	- Series composition	228
26.11.6.	<a href="#">SU_C_Union</a>	- Composition of SUMaPs using the union of the results.	228
26.11.7.	<a href="#">SU_C_ITransform</a>	- Transforms the implementations of a SUMaP.	228
26.11.8.	<a href="#">SU_C_RefineDomain</a>	- Refines the domain of another SUMaP	229
26.11.9.	<a href="#">SU_C_Trace</a>	- Trace of a SUMaP.	229
26.11.10.	<a href="#">SU_C_WrapUnits</a>	- Decorates with units another SUMaP.	229
26.11.11.	<a href="#">SU_L_Exact</a>	- Lifts a UMaP to a SUMaP.	229
26.11.12.	<a href="#">SU_L_Explicit_Approx</a>	- Construct a SUMaP from explicit optimistic and pessimistic approximations.	230
26.11.13.	<a href="#">SU_L_Lift1_Constant</a>	- Lifts a SU1MaP to SUMaP with a constant implementation.	230
26.11.14.	<a href="#">SU_L_Lift1_Transform</a>	- Lifts a SU1MaP to SUMaP by generating the implementations.	230
26.12.	<a href="#">DP</a>	- Design problem with implementations (DPI)	231
26.12.1.	<a href="#">DP_GenericConstant</a>	- A DP with exactly one implementation.	231
26.12.2.	<a href="#">DP_Identity</a>	- The identity design problem.	232
26.12.3.	<a href="#">DP_True</a>	- The DP that is always true.	232
26.12.4.	<a href="#">DP_False</a>	- The DP that is always false.	232
26.12.5.	<a href="#">DP_AmbientConversion</a>	- Compares functionality and resources in an ambient poset.	233
26.12.6.	<a href="#">DP_Catalog</a>	- A DP defined explicitly by a set of options.	233
26.12.7.	<a href="#">DP_Iso</a>	- Enforces isomorphism between functionalities and requirements.	234
26.12.8.	<a href="#">DP_LiftL</a>	- A DP generated from a monotone map from requirements to functionalities.	234
26.12.9.	<a href="#">DP_LiftU</a>	- A DP generated from a monotone map from functionality to requirements.	234
26.12.10.	<a href="#">DP_C_Parallel</a>	- Monoidal product of design problems.	234
26.12.11.	<a href="#">DP_C_Series</a>	- Series composition of DPs.	235
26.12.12.	<a href="#">DP_C_Intersection</a>	- Intersection of design problems	235
26.12.13.	<a href="#">DP_C_Union</a>	- Union of design problems (DPs).	235
26.12.14.	<a href="#">DP_C_Trace</a>	- Trace of a design problem.	235
26.12.15.	<a href="#">DP_FuncNotMoreThan</a>	- Identity with limit to the functionality.	236
26.12.16.	<a href="#">DP_ResNotLessThan</a>	- Identity with limit to the resource.	236
26.12.17.	<a href="#">DP_All_Fi_Leq_R</a>	- Compares a vector of functions to a resource (conjunction).	236
26.12.18.	<a href="#">DP_Any_Fi_Leq_R</a>	- Compares a vector of functions to a resource (disjunction).	236
26.12.19.	<a href="#">DP_F_Leq_All_Ri</a>	- Compares a vector of resources to a function (conjunction).	237
26.12.20.	<a href="#">DP_F_Leq_Any_Ri</a>	- Compares a vector of resources to a function (disjunction).	237
26.12.21.	<a href="#">DP_All_Constants_Leq_R</a>	- Compare a resource to a set of constants	237
26.12.22.	<a href="#">DP_F_Leq_All_Constants</a>	- Compare a functionality to a set of constants	238
26.12.23.	<a href="#">DP_All_Constants_And_F_Leq_R</a>	- Compares resources to a function and a set of constants (conjunction).	238
26.12.24.	<a href="#">DP_Any_Constants_Or_F_Leq_R</a>	- Compares resources to a function and a set of constants (disjunction).	238
26.12.25.	<a href="#">DP_F_Leq_All_R_And_Constants</a>	- Compares a functionality to a resource and a set of constants (conjunction).	239
26.12.26.	<a href="#">DP_F_Leq_Any_R_And_Constants</a>	- Compares a functionality to a resource and a set of constants (disjunction).	239
26.12.27.	<a href="#">DP_C_ExplicitApprox</a>	- Multi-resolution DP	240
26.12.28.	<a href="#">DP_Compiled</a>	- An "opaque" DP defined explicitly by its interface.	240
26.12.29.	<a href="#">DP_Unknown</a>	- Placeholder for an unknown design problem.	240
26.13.	<a href="#">NDP</a>	- Named DPs represent a graph of DPs with named nodes and node ports.	241
26.13.1.	<a href="#">NDP_Composite</a>	- Graph of NDPs with connections between them.	241
26.13.2.	<a href="#">NDP_Simple</a>	- An NDP that contains a single DP.	242
26.13.3.	<a href="#">NDP_Sum</a>	- sum of NDPs	242
26.13.4.	<a href="#">NDP_TemplateHole</a>	- A special NDP to indicate a template hole in the NDP.	243
26.14.	<a href="#">NDPInterface</a>	- The interface of a named DP.	244
26.14.1.	<a href="#">NDPInterface_Explicit</a>	- The interface of a named DP, given by two dictionaries for functionalities and resources.	244
26.15.	<a href="#">NDPTemplate</a>	- A template for an NDP.	245
26.15.1.	<a href="#">NDPTemplate_Simple</a>	- A template described by a graph with holes.	245

26.16.	Query - Queries . . . . .	246
26.16.1.	Query_Single - Single query . . . . .	246
26.17.	Value - A typed value . . . . .	247
26.17.1.	VU - A (poset, value) pair. . . . .	247
26.18.	Check - Checks for the maps, as used in test cases. . . . .	248
26.18.1.	L1Check - Check for a L1Map. . . . .	248
26.18.2.	LCheck - Check for a LMap. . . . .	248
26.18.3.	MapCheck - Check for a monotone map. . . . .	249
26.18.4.	SL1Check - Check for a SL1Map. . . . .	249
26.18.5.	SLCheck - Check for a SL1Map. . . . .	249
26.18.6.	SU1Check - Check for a SU1Map. . . . .	250
26.18.7.	SUCheck - Check for a SUMap. . . . .	250
26.18.8.	U1Check - Check for a U1Map. . . . .	250
26.18.9.	UCheck - Check for a UMap. . . . .	251
<b>27.</b>	<b>Miscellaneous level . . . . .</b>	<b>252</b>
27.1.	LowerSet - Represents a lower set in a poset. . . . .	253
27.1.1.	LowerSet_LowerClosure - A lower set defined as the down closure of a finite set of points. . . . .	253
27.2.	Range - Description of a range of integers. . . . .	254
27.3.	Unit - Units specifications . . . . .	255
27.3.1.	Unit_None - Represents the absence of units. . . . .	255
27.3.2.	Unit_Single - A simple unit. . . . .	255
27.3.3.	Unit_Vector - A vector of units for a product of posets. . . . .	255
27.3.4.	Unit_Wrapped - A special type of unit that is used to describe the units of composite types. . . . .	255
27.4.	UpperSet - Upper sets . . . . .	256
27.4.1.	UpperSet_UpperClosure - An upper set defined as the up closure of a finite set of points. . . . .	256
27.5.	Address - Specifies the origin of an object from a repo and a library. . . . .	257



Part A.

Invitation to computational co-design

# 1. A tour of MCDPL

This chapter provides a tutorial on the language MCDPL and related tools.

## 1.1. What MCDPL is

MCDPL is a *modeling language* that can be used to formally describe *monotone* co-design problems. This means that MCDPL allows describing variables and systems of constraints between variables. MCDPL is not a generic *programming* language. It is not possible to write loops or conditional statements.

MCDPL is designed to represent all and only MCDPs (Monotone Co-Design Problems). This means that variables belong to partially ordered sets, and all relations are monotone. For example, multiplying by a negative number is a syntax error.

MCDPs can be interconnected and composed hierarchically. Several features help organize the models in reusable “libraries”.

Once an MCDP model is described, then we can *query* it in various ways. Another way to see this is that an MCDP model is not a single optimization problem, but rather a collection of optimization problems.

This chapter describes the MCDPL modeling language by way of a tutorial. A more formal description is given in Part B.

## 1.2. Graphical representations of design problems

Monotone design problems are graphically represented as in Fig. 1.

A design problem is represented by a box with curved corners. On the left, solid green arrows represent functionalities; on the right, dashed red arrows represent requirements.

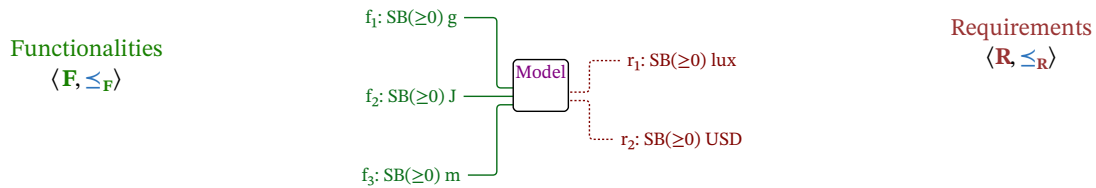


Figure 1.: Representation of a design problem with three functionalities ( $f_1$ ,  $f_2$ ,  $f_3$ ) and two requirements ( $r_1$ ,  $r_2$ ). In this case, the functionality space  $\mathbf{F}$  is the product of three copies of  $\mathbb{R}_{\geq 0}$ :  $\mathbf{F} = \mathbb{R}_{\geq 0}[g] \times \mathbb{R}_{\geq 0}[J] \times \mathbb{R}_{\geq 0}[m]$  and  $\mathbf{R} = \mathbb{R}_{\geq 0}[\text{lux}] \times \mathbb{R}_{\geq 0}[\text{USD}]$ .

The graphical representation of a co-design problem is as a set of design problems that are interconnected (Fig. 2). A functionality and a requirement edge can be joined using a  $\leq$  sign. This is called a “co-design constraint”.

In the figure below, there are two subproblems called  $a$  and  $b$ , and the co-design constraints are  $r_1 \leq f_3$  and  $r_3 \leq f_2$ .

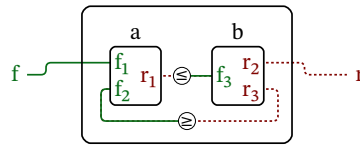


Figure 2.: Example of a *co-design diagram* with two design problems,  $a$  and  $b$ . The *co-design constraints* are  $r_1 \leq f_3$  and  $r_3 \leq f_2$ .

## 1.3. Your first model

An MCDP is described using the keyword `mcdp`. The simplest MCDP is shown in Listing 1. In this case, the body is empty, and that means that there are no functionality and no requirements.

The representation of this MCDP is as in Fig. 3: a simple box with no signals in or out.

Listing 1: empty.mcdp

```
mcdp {  
}
```

Figure 3.



## Adding functionality and requirements

The functionality and requirements of an MCDP are defined using the keywords `provides` and `requires`.

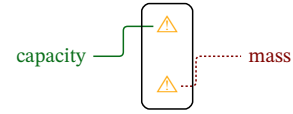
The code in Listing 2 defines an MCDP with one functionality, `capacity`, measured in joules, and one requirement, `mass`, measured in grams.

The graphical representation is in Fig. 4.

Listing 2: model1.mcdp

```
mcdp {  
  provides capacity [J]  
  requires mass [g]  
}
```

Figure 4.



That is, the functionality space is  $\mathbf{F} = \overline{\mathbb{R}}_{\geq 0}[\text{J}]$  and the resource space is  $\mathbf{R} = \overline{\mathbb{R}}_{\geq 0}[\text{g}]$ . Here, let  $\overline{\mathbb{R}}_{\geq 0}[\text{g}]$  refer to the nonnegative real numbers with units of grams.

The MCDP defined above is, however, incomplete, because we have not specified how `capacity` and `mass` relate to one another. In the graphical notation, the co-design diagram has unconnected arrows (Fig. 4).

### 1.3.1. Constraining functionality and requirements

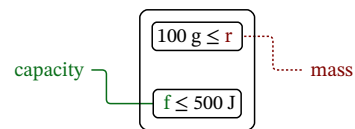
In the body of the `mcdp` declaration one can refer to the values of the functionality and requirements using the expressions `provided functionality name` and `required requirement name`. For example, Listing 3 shows the completion of the previous MCDP, with hard bounds given to both `capacity` and `mass`.

The visualization of these constraints is as in Fig. 5.

Listing 3

```
mcdp {  
  provides capacity [J]  
  requires mass [g]  
  provided capacity ≤ 500 J  
  required mass ≥ 100 g  
}
```

Figure 5.



It is possible to query this minimal example. For example, we can ask through the command-line interface to solve for the minimal requirements given the functionality constraint of 400 J, using the command:

```
| mcdp co-design solve minimal "400 J"
```

The answer is:

```
Minimal resources needed: mass = ↑ {100 g}
```

If we ask for more than the MCDP can provide:

```
| mcdp co-design minimal "600 J"
```

we obtain no solutions: this problem is unfeasible.

### 1.3.2. Use of Unicode glyphs in the language

To describe the inequality constraints, MCDPL allows to use “<=”, “>=”, as well as the Unicode versions “≤”, “≥”, “≤”, “≥”. These two snippets are equivalent:

*using ASCII characters*

```
provided capacity <= 500 J
required mass >= 100g
```

*using Unicode characters*

```
provided capacity ≤ 500 J
required mass ≥ 100g
```

MCDPL also allows to use some special letters in identifiers, such as Greek letters and subscripts.

These two snippets are equivalent:

*using ASCII characters*

```
alpha_1 = beta^3 + 9.81 m/s^2
```

*using Unicode characters*

```
α1 = β3 + 9.81 m/s2
```

### 1.3.3. Aside: a helpful compiler

The MCDPL compiler tries to be very helpful by being liberal in what it accepts and suggests fixes to common mistakes.

For example, if one forgets the keyword `provided`, the compiler will give the following warning:

Please use "provided capacity" rather than just "capacity".

```
line 2 | provides capacity [J]
line 3 | requires mass [g]
line 4 |
line 5 | capacity <= 500 J
.....| ^^^^^^^^
```

The web IDE will automatically insert the keyword using the “beautify” function.

All inequalities are of the kind:

$$\text{resources} \leq \text{functionality}. \quad (1)$$

If you mistakenly put functionality and requirements on the wrong side of the inequality, as in:

```
provided capacity ≥ 500 J # incorrect expression
```

then the compiler will display an error like the following:

```
DPSemanticError: This constraint is invalid.
Both sides are requirements.
line 5 | provided capacity <= 500 J
-----^
```

## 1.4. Describing relations between functionality and resources

In MCDPs, functionality and requirements can depend on each other using any monotone relations.

MCDPL implements some primitive relations, such as addition, multiplication, and exponentiation by a constant.

For example, we can describe a linear relation between mass and capacity, given by the specific energy  $\rho$ :

$$\text{capacity} = \rho \times \text{mass}. \quad (2)$$

This relation can be described in MCDPL as either of the following:

```
provided capacity ≤ ρ · required mass
```

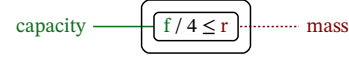
```
provided capacity / ρ ≤ required mass
```

In the graphical representation (Fig. 6), there is now a connection between `capacity` and `mass`, with a DP that multiplies by the inverse of the specific energy.

Listing 4: model14.mcdp

```
mcdp {
  provides capacity [J]
  requires mass [g]
  ρ = 4 J / g
  required mass ≥ provided capacity / ρ
}
```

Figure 6.



For example, if we ask for 600 J:

```
| mcdp co-design solve linear "600 J"
```

we obtain this answer:

```
Minimal resources needed: mass = ↑{150 g}
```

### 1.4.1. Units

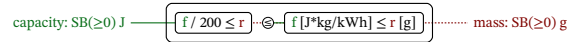
Units are taken seriously. The compiler will complain if there is any dimensionality inconsistency in the expressions. However, as long as the dimensionality is correct, it will automatically convert to and from equivalent units.

For example, in Listing 5 the specific energy is given in kWh/kg. PyMCDP will take care of the needed conversions, and will introduce a conversion from J\*kg/kWh to g (Fig. 7).

Listing 5: model15.mcdp

```
mcdp {
  ''' Model of a battery '''
  provides capacity [J]
  requires mass [g]
  ρ = 200 kWh / kg 'Specific energy (kWh/kg)'
  required mass ≥ provided capacity / ρ
}
```

Figure 7.



Listing 5 also shows the syntax for comments. MCDPL allows adding a Python-style documentation string at the beginning of the model, delimited by three quotes. It also allows giving a short description for each functionality, requirement, or constant declaration, by writing a Python-style string at the end of the line.

## 1.5. Catalogs

The previous example used a linear relation between functionality and requirement. However, in general, MCDPs do not make any assumption about continuity and differentiability of the functionality-requirement relation. The MCDPL language has a construct called [catalog](#) that allows defining an arbitrary discrete relation.

Recall from the theory that a design problem is defined from a triplet of [functionality space](#), [implementation space](#), and [requirement space](#). According to the diagram in Fig. 8, one should define the two maps [req](#) and [prov](#), which map an implementation to the functionality it provides and the requirements it requires.

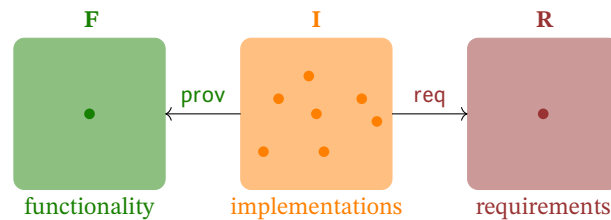


Figure 8.: A design problem is defined from an implementation space **I**, a functionality space **F**, a requirement space **R**, and the maps [req](#) and [prov](#) that relate the three spaces.

MCDPL allows defining arbitrary maps [req](#) and [prov](#), and therefore arbitrary relations from functionality to requirements, using the [catalog](#) construction.

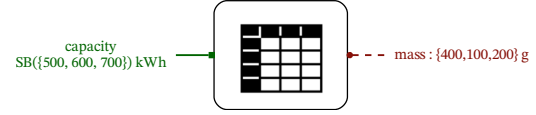
An example is shown in Listing 6. In this case, the implementation space contains the three elements `model1`, `model2`, `model3`. Each model is explicitly associated with a value in the functionality and the requirements space.

The icon for this construction is meant to remind us of a spreadsheet (Fig. 9).

Listing 6: `catalog1.mcdp`

```
catalog {
  provides capacity [kWh]
  requires mass [g]
  500 kWh ↔ model1 ↔ 100 g
  600 kWh ↔ model2 ↔ 200 g
  700 kWh ↔ model3 ↔ 400 g
}
```

Figure 9.



In Fig. 9 we can see how the type of the mass is not just grams, but rather given as `{100, 200, 400 g}`. The solver propagates the bounds information about functionalities and requirements and uses this information to prune the search space during querying.

### 1.5.1. Multiple minimal solutions

The `catalog` construct is the first construct we encountered that allows defining MCDPs that have *multiple minimal solutions*. To see this, let's expand the model in Listing 6 to include a few more models and one more requirement, `cost`.

Listing 7: `catalog2.mcdp`

```
catalog {
  provides capacity [kWh]
  requires mass [g]
  requires cost [USD]

  500 kWh ↔ model1 ↔ 100 g, 10 USD
  600 kWh ↔ model2 ↔ 200 g, 200 USD
  600 kWh ↔ model3 ↔ 250 g, 150 USD
  700 kWh ↔ model4 ↔ 400 g, 400 USD
}
```

The numbers (not realistic) were chosen so that `model2` and `model3` do not dominate each other: they provide the same functionality (`600 kWh`) but one is cheaper but heavier, and the other is more expensive but lighter. This means that for the functionality value of `600 kWh` there are two minimal solutions: either `<200 g, 200 USD>` or `<250 g, 150 USD>`.

The number of minimal solutions is not constant: for this example, we have four cases as a function of  $f$  (Table 1.1). As  $f$  increases, there are 1, 2, 1, and 0 minimal solutions.

Table 1.1.: Solution cases for the model in Listing 7.

Functionality required	Optimal implementation(s)	Minimal resources needed
$0 \text{ kWh} \leq f \leq 500 \text{ kWh}$	<code>model1</code>	<code>&lt;100 g, 10 USD&gt;</code>
$500 \text{ kWh} < f \leq 600 \text{ kWh}$	<code>model2</code> OR <code>model3</code>	<code>&lt;200 g, 200 USD&gt;</code> OR <code>&lt;250 g, 150 USD&gt;</code>
$600 \text{ kWh} < f \leq 700 \text{ kWh}$	<code>model4</code>	<code>&lt;400 g, 400 USD&gt;</code>
$700 \text{ kWh} < f \leq \text{Top kWh}$	(unfeasible)	$\emptyset$

We can verify these with `mcdp co-design solve`. We also use the switch `--imp` to ask the program to give the name of the implementations; without the switch, it only prints the value of the minimal resources.

For example, for  $f = 50 \text{ kWh}$ :

```
mcdp co-design solve --imp catalog2 "50 kWh"
```

we obtain one solution:

```
Minimal resources needed: mass, cost = ↑{mass:100 g, cost:10 USD}
r = <mass:100 g, cost:10 USD>
implementation 1 of 1: m = 'model1'
```

For  $f = 550$  kWh:

```
mcdp co-design solve --imp catalog2 "550 kWh"
```

we obtain two solutions:

```
Minimal resources needed:
mass, cost = ↑{mass:200 g, cost:200 USD}, {mass:250 g, cost:150 USD}

r = <mass:250 g, cost:150 USD>
implementation 1 of 1: m = 'model3'

r = <mass:200 g, cost:200 USD>
implementation 1 of 1: m = 'model2'
```

The solver displays first the set of minimal resources required; then, for each value of the resource, it displays the name of the implementations; in general, there could be multiple implementations that have the same resource consumption.

## 1.6. Union/choice of design problems

The “choice” construct allows describing the idea of “alternatives”.

As an example, let us consider how to model the choice between different battery technologies.

Consider the model of a battery, in which we take the functionality to be the **capacity** and the requirements to be the **mass [g]** and the **cost [USD]**.

Consider two different battery technologies, characterized by their specific energy (**Wh/kg**) and specific cost (**Wh/USD**).

Specifically, consider Nickel-Hydrogen batteries (NiH2) and Lithium-Polymer (LiPo) batteries. One technology is cheaper but leads to heavier batteries and vice versa. Because of this fact, there might be designs in which we prefer either.

First, we model the two battery technologies separately as two MCDPs that have the same interface (same requirements and same functionalities).

Listing 8: Battery1\_LiPo.mcdp

```
mcdp {
  provides capacity [J]
  requires mass [g]
  requires cost [USD]
  ρ = 150 Wh/kg
  α = 2.50 Wh/USD
  required mass ≥ provided capacity / ρ
  required cost ≥ provided capacity / α
}
```

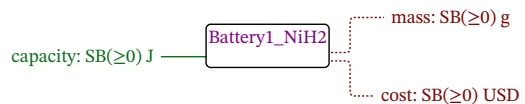
Listing 9: Battery1\_NiH2.mcdp

```
mcdp {
  provides capacity [J]
  requires mass [g]
  requires cost [USD]
  ρ = 45 Wh/kg
  α = 10.50 Wh/USD
  required mass ≥ provided capacity / ρ
  required cost ≥ provided capacity / α
}
```

Figure 10.



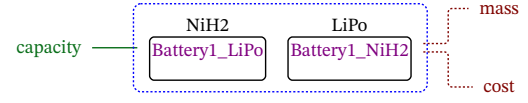
Figure 11.



Listing 10: Batteries.mcdp

```
choose(
  NiH2: `Battery1_LiPo,
  LiPo: `Battery1_NiH2
)
```

Figure 12.



## 1.7. Composing design problems

The MCDPL language encourages composition and code reuse through composition of design problems.

### 1.7.1. Implicit composition of design problems from formulas

All the mathematical operations (addition, multiplication, power, etc.) are each represented by their own design problem, even though they do not have explicit names.

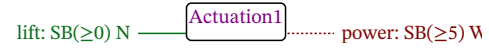
For example, let's define the MCDP `Actuation1` to represent the actuation in a drone. We model it as a quadratic relation from `lift` to `power`, as in Listing 11.

Listing 11: Actuation1.mcdp

```
mcdp {
  provides lift [N]
  requires power [W]

  l = provided lift
  p0 = 5 W
  p1 = 6 W/N
  p2 = 7 W/N²
  required power ≥ p0 + p1 · l + p2 · l²
}
```

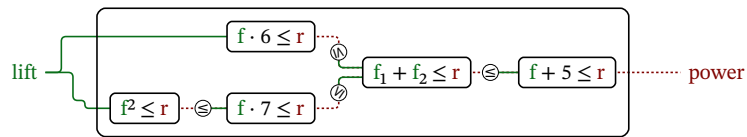
Figure 13.



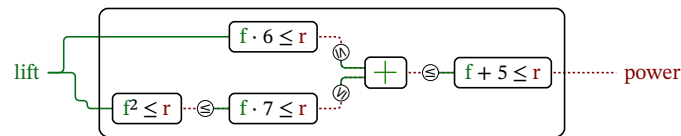
Note that the relation between `lift` and `power` is described by the polynomial relation

$$\text{required power} \geq p_0 + p_1 \cdot l + p_2 \cdot l^2$$

This relation can be written as the composition of several DPs, corresponding to sum, multiplication, and exponentiation (Fig. 13).



In the following, we will use icons for the operations:



### 1.7.2. Explicit composition of design problems

The other way to compose design problems is for the user to define them separately and then connect them explicitly.

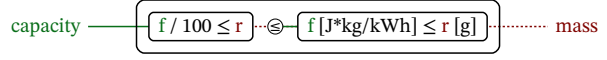
Suppose we define a model called `Battery` as in Fig. 14.



Listing 12: Battery1.mcdp

```
mcdp {
  provides capacity [J]
  requires mass [g]
  ρ = 100 kWh / kg # specific_energy
  required mass ≥ provided capacity / ρ
}
```

Figure 14.

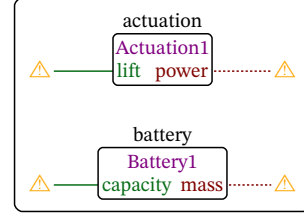


We can now put the battery model and the actuation model using the following syntax.

Listing 13: combined1.mcdp

```
mcdp {
  sub actuation = instance `Actuation1
  sub battery = instance `Battery1
}
```

Figure 15.



The syntax to re-use previously defined MCDPs is `instance `Name`. The backtick means “load the MCDP from the library, from the file called `Name.mcdp`”.

Simply instantiating the models puts them side-to-side in the same box; we need to connect them explicitly. The model in Listing 13 is not usable yet because some of the edges are unconnected (Fig. 15). We can create a complete model by adding co-design constraints.

### 1.7.3. Adding co-design constraints

For example, suppose that we know the desired `endurance` for the design. Then we know that the `capacity` provided by the battery must exceed the `energy` required by actuation, which is the product of power and endurance. All of this can be expressed directly in MCDPL using the syntax:

```
energy = provided endurance · (power required by actuation)
capacity provided by battery ≥ energy
```

The visualization of the resulting model has a connection between the two design problems representing the co-design constraint (Listing 14).

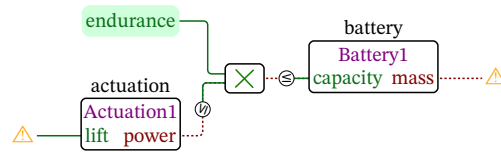
Listing 14: combined2.mcdp

```
mcdp {
  provides endurance [s]

  sub actuation = instance `Actuation1
  sub battery = instance `Battery1

  # battery must provide power for actuation
  energy = provided endurance · power required by actuation
  capacity provided by battery ≥ energy
  # still incomplete...
}
```

Figure 16.



We can create a model with a loop by introducing another constraint.

Take `payload` to represent the user payload that we must carry.

Then the lift provided by the actuator must be at least the mass of the battery plus the mass of the payload times gravity:

```
constant gravity = 9.81 m/s²
total_mass = (mass required by battery + provided payload)
weight = total_mass · gravity
lift provided by actuation ≥ weight
```

Now there is a loop in the co-design diagram (Fig. 17).

Listing 15: Composition.mcdp

```
from library . import model Battery1, Actuation1
mcdp {
    provides endurance [s]
    provides payload [g]

    sub actuation = instance Actuation1
    sub battery = instance Battery1

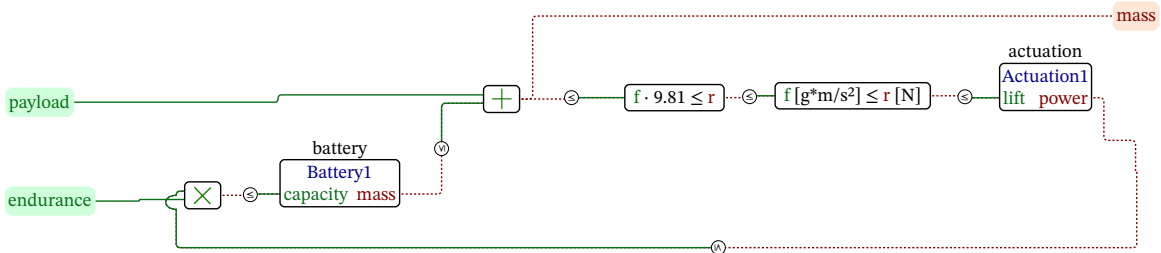
    # battery must provide power for actuation
    energy = provided endurance * (power required by actuation)

    capacity provided by battery ≥ energy

    # actuation must carry payload + battery
    constant gravity = 9.81 m/s2
    total_mass = (mass required by battery + provided payload)
    weight = total_mass * gravity
    lift provided by actuation ≥ weight

    # minimize total mass
    requires mass [g]
    required mass ≥ total_mass
}
```

Figure 17.



## 1.8. Re-usable design patterns using templates

“Templates” are a way to describe reusable design patterns.

For example, the code in Listing 15 composes a particular battery model, called `Battery1`, and a particular actuator model, called `Actuation1`. However, it is clear that the pattern *interconnect battery and actuators* is independent of the particular battery and actuator selected. MCDPL allows describing this situation by using “templates”.

Templates are contained in files with extension `.mcdp_template`. The syntax is:

```
template [name1: interface1, name2: interface2]
mcdp {
    # usual definition here
}
```

In the brackets put pairs of names and NDPs that will be used to specify the interface. Interfaces are contained in files with extension `.mcdp_interface`. For example, Fig. 18 defines an interface with a functionality and a requirement.

Then we can declare a template as in Listing 16. The template is visualized as a diagram with a hole (Fig. 19).

Figure 18.

ExampleInterface.mcdp\_interface

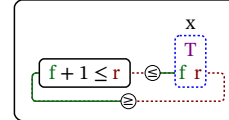
```
interface {
  provides f [N]
  requires r [N]
}
```

Listing 16

ExampleTemplate.mcdp\_template

```
template [T: `ExampleInterface]
mcdp {
  sub x = instance T
  f provided by x ≥ r required by x + 1
}
```

Figure 19.



## Example

Here is the application to the previous example of battery and actuation. Suppose that we define their interfaces as in Listing 17 and Listing 18.

Listing 17: BatteryInterface.mcdp\_interface

```
interface {
  provides capacity [J]
  requires mass [g]
}
```

Listing 18: ActuationInterface.mcdp\_interface

```
interface {
  provides lift [N]
  requires power [W]
}
```

Then we can define a template that uses them. For example the code in Listing 19 specifies that the templates requires two parameters, called `generic_actuation` and `generic_battery`, and they must have the interfaces defined by ``ActuationInterface` and ``BatteryInterface`.

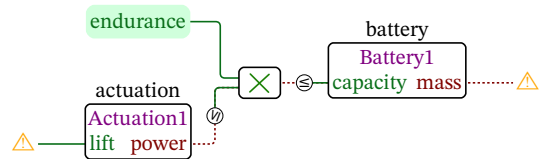
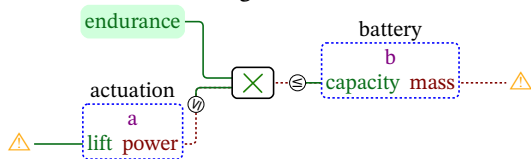
Listing 19: CompositionTemplate.mcdp\_template

```
template [
  a: `ActuationInterface,
  b: `BatteryInterface
]
mcdp {
  sub actuation = instance a
  sub battery = instance b
  # battery must provide power for actuation
  provides endurance [s]
  energy = provided endurance · (power required by actuation)
  capacity provided by battery ≥ energy
  # only partial code
  # ...
}
```

Listing 20: TemplateUse.mcdp

```
specialize [
  a: `Actuation1,
  b: `Battery1
] `CompositionTemplate
```

Figure 20.



The diagram in Fig. 20 has two “holes” in which we can plug any compatible design problem. To fill the holes with the models previously defined, we can use the keyword “`specialize`” as in Listing 20.

Part B.

MCDPL reference

## 2. Introduction

### 2.1. Introduction

#### 2.1.1. Kinds

MCDPL has 7 *kinds* of data:

1. *Posets*;
2. *Values* that belong to a poset;
3. *Intervals* of values, also called *uncertain values*;
4. *Primitive DPs*: corresponding to the morphisms in the category **DP**. These are the basic building blocks of the language but they are not used directly by the user.
5. *Named DPs (NDPs)*: these are DPs with functionality and requirements begin tuples of posets with names associated to them. These can be:
  - *atomic*, often a simple wrapping of a primitive DP.
  - *composite*, which are defined as the interconnection of other NDPs. These are also called *Monotone Co-Design Problems (MCDPs)*.
6. *Interfaces*: these are the types of the ports of an NDP.
7. *Templates*: These are *diagrams with holes* that can be *specialized* to create NDPs.

Table 2.1 shows the various kinds and examples of expressions belonging to them. The *extension* column shows the file extension used for files containing expressions of that kind.

Table 2.1.: *Kinds* in MCDPL

kind	mathematical concept	extension	code snippet example
<i>Posets</i>	(sub)posets	.mcdp_poset	<code>product(mass: g, volume: m<sup>3</sup>)</code>
<i>Values</i>	elements of posets	.mcdp_value	<code>{10 g, 20 l}</code>
Uncertain values	intervals of values	n/a	<code>between 10 g and 12 g</code>
<i>Primitive DPs</i>	morphisms of <b>DP</b>	.mcdp_primitivedp	<code>yaml resource('dpc1.dpc.yaml')</code>
<i>MCDPs</i>	diagrams of <b>DP</b> with signal names	.mcdp	<pre>mcdp {   requires r = 10 g }</pre>
<i>Interfaces</i>	hom-sets of <b>DP</b> with signal names	.mcdp_interfaces	<pre>interface {   requires r [g]   provides f [W] }</pre>
Templates	operad of <b>DP</b> with signal names	.mcdp_template	<pre>template [] mcdp { }</pre>

## 3. Posets and values

### 3.1. `poset`: Defining finite posets

It is possible to define custom finite posets in files `*.mcdp_poset` using the keyword `poset`.

For example, to define the poset  $Q = \langle Q, \leq_Q \rangle$  with elements

$$Q = \{a, b, c, d, e, f, g\} \quad (1)$$

and the order relations

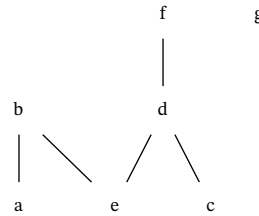
$$a \leq_Q b, \quad c \leq_Q d, \quad e \leq_Q d \leq_Q f, \quad e \leq_Q b, \quad (2)$$

we create a file named `q.mcdp_poset` with code as in Listing 21.

Listing 21: `q.mcdp_poset`

```
poset {
  a ≤ b
  c ≤ d
  e ≤ d ≤ f
  e ≤ b
  g
}
```

Figure 1.: Hasse diagram of the poset defined in Listing 21



After the poset has been defined, it can be used in the definition of an MCDP, by referring to it by name using the backtick notation, as in “``Q`”.

To refer to its elements, use the notation ``Q: element` (Listing 22).

Listing 22: `onep.mcdp_poset`

```
mcdp {
  provides f [`Q]
  provided f ≤ `Q: d
}
```

Figure 2.



### 3.2. Extrema of posets

#### 3.2.1. Top and Bottom

To indicate top and bottom of a poset, use this syntax:

`Top poset`  
`Bottom poset`

`⊤ poset`  
`⊥ poset`

For example, `Top V` indicates the top of the poset `V`.

#### 3.2.2. Minimals and Maximals

The expressions `Minimals poset` and `Maximals poset` denote the set of minimal and maximal elements of a poset.

For example, assume that the poset `MyPoset` is defined as in Fig. 1. Then the expression `Maximals `MyPoset` is equivalent to the set with two elements `b` and `d`, and the expression `Minimals `MyPoset` is equivalent to a set with the elements `a`, `e`, `c`.

These assertions can be checked with the following code:

```
assert_equal(Maximals `MyPoset, {`MyPoset: b, `MyPoset: d})
```

```
assert_equal(Minimals `MyPoset, {`MyPoset: a, `MyPoset: e, `MyPoset: c})
```

### 3.3. Numerical posets

MCDPL works with numerical posets that are models of the completion of  $\mathbb{R}$ , with  $-\infty$  and  $+\infty$  as bottom and top.

These subposets are parametrized by:

1. A lower bound;
2. An upper bound;
3. A discretization step (possibly 0, to mean no discretization).
4. A unit.

The keywords `Nat`, `Int`, and `dimensionless` refer to a model of  $\mathbb{N}$ ,  $\mathbb{Z}$ , and  $\mathbb{R}$  respectively (unitless).

poset	lower bound	upper bound	discretization
<code>Nat</code>	0	$+\infty$	1
<code>Int</code>	$-\infty$	$+\infty$	1
<code>dimensionless</code>	0	$+\infty$	0

The compiler will derive the step and lower and upper bounds of expressions using abstract interpretation of the mathematical operations.

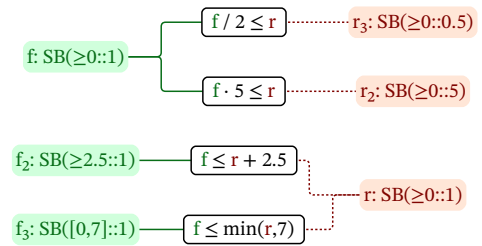
Listing 23 shows an example in which some functionality/resource is defined as a natural number, and other derived quantities are inferred to be in subposets.

Listing 23: `simplenumerics.mcdp`

```
mcdp {
  requires r [Nat]
  provides f [Nat]

  provides f_2 = required r + 2.5
  provides f_3 = min(required r, 7)
  requires r_2 = provided f * 5
  requires r_3 = provided f / 2
}
```

Figure 3.



#### 3.3.1. Numerical precision

The details of how these posets are represented internally for computation is considered an *implementation detail*.

By default, numbers are represented as *decimal values* with 9 digits of precision.

Note that monotone co-design theory does not require that the operations are exact; in particular, it does not require that the operations are associative or commutative. Therefore, some of the usual concerns of numerical analysis do not apply.

It is not a problem if  $a + b \neq b + a$ . What matters is that the operations are *correct* in the sense that they are *conservative*. For example, the operation  $a + b$  operating with finite precision needs to be rounded *up* if we are computing a feasible *upper set*, and rounded down if we are computing a feasible *lower set*, as those are *conservative* choices.

### 3.4. Numbers with units

Numerical values can also have *units* associated to them. So we can distinguish  $\overline{\mathbb{R}}_{\geq 0}[\text{m}]$  from  $\overline{\mathbb{R}}_{\geq 0}[\text{s}]$  and even  $\overline{\mathbb{R}}_{\geq 0}[\text{m}]$  from  $\overline{\mathbb{R}}_{\geq 0}[\text{km}]$ .

These posets and their values are indicated using the syntax in Table 3.1.

In general, units behave as one might expect. Units are implemented using the library `Pint`<sup>1</sup>; please see its documentation for more information.

<sup>1</sup><http://pint.readthedocs.org/>

Table 3.1.: Numbers with units

syntax for posets	dimensionless	g	J	m/s
syntax for values	23	1.2 g	20 J	23 m/s

The following is the formal definition of the operations involving units.

Units form a group with an equivalence relation.

Call this group of units  $U$  and its elements  $u, v, \dots \in U$ . By  $\mathbb{F}[u]$ , we mean a field  $\mathbb{F}$  enriched with an annotation of units  $u \in U$ .

Multiplication is defined for all pairs of units. Let  $|x|$  denote the absolute numerical value of  $x$  (stripping the unit away). Then, if  $x \in \mathbb{F}[u]$  and  $y \in \mathbb{F}[v]$ , their product is  $x \cdot y \in \mathbb{F}[uv]$  and  $|x \cdot y| = |x| \cdot |y|$ .

Addition is defined only for compatible pairs of units (e.g.,  $m$  and  $km$ ), but it is not possible to sum, say,  $m$  and  $s$ . If  $x \in \mathbb{F}[u]$  and  $y \in \mathbb{F}[v]$ , then  $x + y \in \mathbb{F}[u]$ , and  $x + y = |x| + \alpha_v^u |y|$ , where  $\alpha_v^u$  is a table of conversion factors, and  $|x|$  is the absolute numerical value of  $x$ .

In practice, this means that MCDPL thinks that  $1 \text{ kg} + 1 \text{ g}$  is equal to  $1.001 \text{ kg}$ . Addition is not strictly commutative, because  $1 \text{ g} + 1 \text{ kg}$  is equal to  $1001 \text{ g}$ , which is equivalent, but not equal, to  $1.001 \text{ kg}$ .

### 3.5. Poset products

MCDPL allows the definition of finite Cartesian products, which can be anonymous or named.

#### 3.5.1. $\times$ : Anonymous Poset Products

Use the Unicode symbol “ $\times$ ” or the simple letter “ $x$ ” to create a poset product, using the syntax:

```
 $P1 \times P2 \times \dots \times Pn$ 
```

For example, the expression  $J \times A$  represents a product of Joules and Amperes.

The elements of a poset product are tuples. To define a tuple, use angular brackets “ $\langle$ ” and “ $\rangle$ ”.

For example, the expression “ $\langle 2 J, 1 A \rangle$ ” denotes a tuple with two elements, equal to  $2 J$  and  $1 A$ . An alternative syntax uses the fancy Unicode brackets “ $\langle$ ” and “ $\rangle$ ”, as in “ $\langle 2 J, 1 A \rangle$ ”.

Tuples can be nested. For example, you can describe a tuple like

```
 $\langle \langle 2 J, 1 A \rangle, \langle 1 m, 1 s, 42 \rangle \rangle,$ 
```

and its poset is denoted as

```
 $(J \times A) \times (m \times s \times \mathbb{N}).$ 
```

#### 3.5.2. `product`: Named Poset Products

MCDPL also supports “named products”. These are semantically equivalent to products, however, there is also a name associated to each entry. This allows to easily refer to the elements. For example, the following declares a product of the two spaces  $J$  and  $A$  with the two entries named “energy” and “current”.

```
product(energy: J, current: A)
```

The names for the fields must be valid identifiers (starts with a letter, contains letters, underscore, and numbers).

The attribute can be referenced using the following syntax:

```
(requirement).field
```

For example:



```
mcdp {
  provides out [product(energy: J, current: A)]

  (provided out).energy ≤ 10 J
  (provided out).current ≤ 2 A
}
```

## 3.6. Posets of subsets

### 3.6.1. powerset: Power sets

MCDPL allows to describe the set of subsets of a poset, i.e. its *power set*.

The syntax is either of these:

`powerset(p)`

`p(P)`

To describe the values in a powerset (subsets), use this set-building notation:

`{value, value, ..., value}`

For example, the value `{1, 2, 3}` is an element of the poset `powerset(Nat)`.

### 3.6.2. EmptySet: The empty set

To denote the empty set, use the keyword `EmptySet` or the equivalent unicode:

`EmptySet P`   `∅ P`

Note that empty sets are typed. This is different from set theory: here, a set of apples without apples and a set of oranges without oranges are two different sets, while in traditional set theory they are the same set.

`∅ J` is an empty set of energies, and `∅ V` is an empty set of voltages, and the two are not equivalent.

### 3.6.3. UpperSets and LowerSets: Upper and lower sets

The poset of upper sets of *P* can be described by the syntax

`UpperSets(P)`

For example, `UpperSets(N)` is the poset of upper sets for the natural numbers.

To describe an upper set (i.e. an element of the space of upper sets), use the keyword `upperclosure` or its abbreviation `↑`. The syntax is:

`upperclosure set`   `↑ set`

For example: `↑{2 g, 1 m}` denotes the principal upper set of `{2 g, 1 m}` in the poset `g x m`.

Similarly you can define the poset of lower sets of *P* as:

`LowerSets(P)`

and you can obtain elements of this poset using the keyword `lowerclosure` or its abbreviation `↓`.

`lowerclosure set`   `↓ set`

### 3.7. Defining uncertain constants

MCDPL allows describing interval uncertainty for variables and expressions.

There are three syntaxes accepted:

1. The first is using explicit bounds:

```
x = between lower bound and upper bound
```

2. Median plus or minus absolute tolerance:

```
x = median +- tolerance    x = median ± tolerance
```

3. Median plus or minus percent:

```
x = median +- percent tolerance %    x = median ± percent tolerance %
```

For example, Table 3.2 shows the different ways in which a constant can be declared to be between 9.95 kg and 10.05 kg :

Table 3.2.: Equivalent ways to declare an uncertain constant

```
x = between 9.95 kg and 10.05 kg
```

```
x = 10 kg ± 50 g
```

```
constant c = 10 kg  
δ = 50 g  
x = between c - δ and c + δ
```

It is also possible to describe parametric uncertain relations by simply using uncertain constants in place of regular constants:

```
mcdp {  
  provides energy [J]  
  requires mass [kg]  
  
  specific_energy = between 100 kWh/kg and 120 kWh/kg  
  required mass * specific_energy ≥ provided energy  
}
```

## 4. Named DPs

### 4.1. Defining NDPs

An NDP (*named* design problem) is a design problem that has associated names for its functionality and requirements.

There are several ways of defining an NDP:

1. by constructing one using the `catalog` syntax;
2. by constructing one using the `mcdp` syntax;
3. by constructing one using the `dp` syntax;
4. by specializing a template, using the `specialize` keyword.
5. by constructing one using operations such as `compact`, `abstract`, etc.

### 4.2. Constructing NDPs as catalogs

An NDP can be created using a “catalog” declaration, which enumerates the possible ways to provide functionalities and requirements explicitly.

A catalog declaration is comprised of zero or more functionality/requirement declarations and zero or more *catalog records*:

```
catalog {  
    provides f1 [W]  
    requires r1 [s]  
  
    # records go here  
}
```

There are two types of records that can be used (but cannot be mixed):

1. Records that specify the implementation names explicitly;
2. Records that do not specify the implementation names implicitly.

```
catalog {  
    provides f1 [W]  
    requires r1 [s]  
  
    10 W  $\leftrightarrow$  imp1  $\rightarrow$  10 s  
    20 W  $\leftrightarrow$  imp2  $\rightarrow$  20 s  
}
```

```
catalog {  
    provides f1 [W]  
    requires r1 [s]  
  
    10 W  $\leftrightarrow$  10 s  
    20 W  $\leftrightarrow$  20 s  
}
```

For multiple functionalities and resources, partition the elements using commas:

```
catalog {  
    provides f1 [W]  
    provides f2 [m]  
    requires r1 [s]  
    requires r2 [s]  
  
    5 W, 5 m  $\leftrightarrow$  imp1  $\rightarrow$  10 s, 10 s  
}
```

Note that in the catalog rows, you must use units, which might be different from the units used in the declaration of the functionalities and requirements. For example, in the following we use standard units (meters and seconds) for the catalog rows, but we use different units in the declaration of the functionalities and requirements (miles and hours):

```
catalog {
  provides distance [m]
  requires duration [s]
  5 miles ↔ 10 hours
}
```

In case there are no functionalities, use an empty tuple on the left; and if there are no requirements, use an empty tuple on the right.

```
catalog {
  requires r1 [s]
  {} ↔ imp1 → 10 s
}
```

```
catalog {
  provides f1 [s]
  5s ↔ imp1 → {}
}
```

#### 4.2.1. True and false

The empty catalog is valid: it is the NDP with no functionality and no requirements and no implementations. That is:

```
catalog {
}
```

This is “false”: even though nothing is asked, there is no way to provide it.

Dually, the following catalogs definitions are equivalent to “true”. One has an anonymous implementation, the other has an explicit implementation.

```
catalog {
  {} ↔ imp1 → {}
}
```

```
catalog {
  {} ↔ {}
}
```

Of course, we can create a catalog that is *even more true*, by adding more implementations:

```
catalog {
  {} ↔ even → {}
  {} ↔ more → {}
  {} ↔ ways → {}
  {} ↔ to → {}
  {} ↔ provide → {}
  {} ↔ nothing → {}
}
```

### 4.3. Constructing NDPs from YAML files

An NDP can be created using a “dp” declaration, which is used to import from external YAML data.

The syntax is as follows:

```
dp {
  # usual provides/requires declaration

  # only one line of this form
  implemented-by yaml resource("data.yaml")
}
```

For example:

```
dp {
  provides task [`task]
  requires sensor_requirements [`sensor_reqs]
  requires vehicle_properties [`vehicle_properties]
  requires computation [ops]
  requires discomfort [dimensionless]

  implemented-by yaml resource("planner_catalog-Mar-12-2024-08-24.dpc.yaml")
}
```

An example YAML file is as follows:

```
F:
- ``task"
R:
- ``sensor_reqs"
- ``vehicle_properties"
- "ops"
- "dimensionless"
implementations:
  planning_4125:
    f_max:
      - ``task: task_urban_car_cr_AA_50_3"
    r_min:
      - ``sensor_reqs: sensor_reqs_4125"
      - ``vehicle_properties: car_van_chrysler_pacifica"
      - "6.00 ops"
      - "13 dimensionless"
  planning_4102:
    f_max:
      - ``task: task_urban_car_cr_AA_30_3"
    r_min:
      - ``sensor_reqs: sensor_reqs_4102"
      - ``vehicle_properties: car_van_chrysler_pacifica"
      - "6.00 ops"
      - "11 dimensionless"
```

The files should contain three fields: F, R, and implementations.

The first two fields are lists of strings. These represent, in MCDPL, the names of the interfaces that the NDP requires and provides.

The third field is a dictionary, where the keys are the names of the implementations, and the values are dictionaries with two keys: `f_max` and `r_min`. These are lists of strings, representing the requirements and capabilities of the implementation in MCDPL.

## 4.4. Describing Monotone Co-Design Problems

An NDP can be created using a declaration enclosed in the `mcdp` construct. The declaration must be comprised of an optional comment and zero or more statements; a statement can be either a declaration or a constraint, and these can be mixed.

### 4.4.1. Declaring functionality and requirements explicitly

A functionality declaration is of this form:

```
provides f [poset]
```

Symmetrically, a requirement declaration is of this form:

```
requires r [poset]
```

#### 4.4.2. Declaring functionality and requirements implicitly using expressions

There are shortcuts one can use to declare functionalities and requirements given a value:

```
provides f ≤ expression
```

This is equivalent to:

```
provides f [poset]
provided f ≤ expression
```

Symmetrically, there are the same constructs for defining requirements:

```
requires r ≥ expression
```

This is equivalent to:

```
requires r [poset]
required r ≥ expression
```

#### 4.4.3. Forwarding functionalities and requirements from other subproblems

The second shortcut is used to declare one or more functionalities and specify which DP is responsible for providing them:

```
provides f1, f2, ... using dp
```

This is equivalent to

```
provides f1 ≤ f1 provided by dp
provides f2 ≤ f2 provided by dp
```

The symmetric construct is used to declare one or more requirements and specify which DP is responsible for providing them:

```
requires r1, r2, ... for dp
```

which is equivalent to:

```
requires r1 ≥ r1 required by dp
requires r2 ≥ r2 required by dp
```

Note that the syntax is “requires ...for ...” rather than “provides ...using ...”.

#### 4.4.4. Declaring functionality and requirements using interfaces

Another way to declare functionalities and requirements is to declare that one is implementing a certain interface.

For example, given the following interface definition:

```
interface {
  provides f [W]
  requires r [g]
}
```

We can use the keyword `implements` to declare that the current MCDP is implementing the interface:

```
mcdp {
  implements `MyInterface

  provided f ≤ 10 W
  required r ≥ 1 kg
}
```

#### 4.4.5. Ignoring or propagating the functionality/requirements of subproblems

In general, the compiler will complain if there are unconnected ports: each functionality or requirement of subproblems must be connected to something: either another subproblem, or to the global functionality or requirement of the parent problem.

Sometimes, it is useful to ignore some of the functionality or requirements of subproblems.

##### `ignore`: Ignoring specific functionality/requirements

The `ignore` keyword can be used to mark some of the functionality or requirements of an NDP as “ignored”. The compiler will then not complain if they are unconnected.

The syntax is:

```
ignore f provided by dp
```

```
ignore r required by dp
```

Internally, the compiler will connect the ignored functionality or requirements to a special DP that is always true.

##### `ignore unconnected`: Ignoring all unconnected

There is a special declaration `ignore unconnected` that can be used to ignore all unconnected ports.

```
ignore unconnected
```

##### `propagate unconnected`: Propagating unconnected ports

Finally, there is a special declaration `propagate unconnected` that can be used to “propagate” unconnected ports:

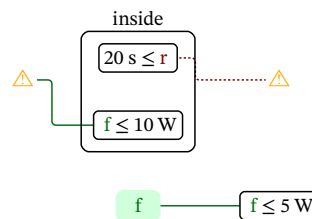
```
propagate unconnected
```

Consider the following example, in which there is a subproblem called `inside` that provides and requires some ports, but these ports are not connected to anything:

```
mcdp {
  provides f ≤ 5 W

  sub inside = instance mcdp {
    provides f_sub ≤ 10 W
    requires r_sub ≥ 20 s
  }

  # f_sub, r_sub are unconnected
}
```



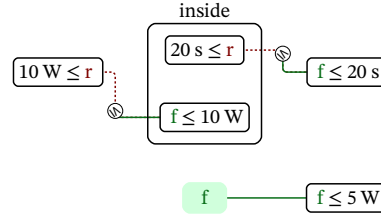
In the web editor, there will be warnings about unconnected ports.

If we add the declaration `ignore unconnected`, then the unconnected ports will be ignored; they will be connected to a special DP that is always true.

```
mcdp {
  provides f ≤ 5 W

  sub inside = instance mcdp {
    provides f_sub ≤ 10 W
    requires r_sub ≥ 20 s
  }

  ignore unconnected
}
```

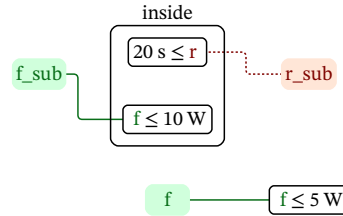


If we add the declaration `propagate unconnected`, then the unconnected ports will be propagated to the parent DP.

```
mcdp {
  provides f ≤ 5 W

  sub inside = instance mcdp {
    provides f_sub ≤ 10 W
    requires r_sub ≥ 20 s
  }

  propagate unconnected
}
```



#### 4.4.6. Shortcuts for summing over functionalities and requirements

One common pattern is to sum over many requirements of the same name. For example, a design might consist of several components, and the budgets must be summed together (Listing 24). In this case, it is possible to use the shortcut `sum` (or its equivalent symbol  $\sum$ ) that allows to sum over requirements required with the same name (Listing 25) with the syntax

sum budget required by \*

An error will be generated if there are no subproblems with a requirement of the given name.

The two MCDP in Listing 24 and Listing 25 are equivalent.

Listing 24: sumc-example.mcdp

```
mcdp {
  requires total_budget [USD]

  sub c1 = instance `Component1
  sub c2 = instance `Component2
  sub c3 = instance `Component3

  required total_budget ≥ (
    budget required by c1 +
    budget required by c2 +
    budget required by c3
  )
}
```

Listing 25: sumc2.mcdp

```
mcdp {
  requires total_budget [USD]

  c1 = instance `Component1
  c2 = instance `Component2
  c3 = instance `Component3

  # this sums over all components
  required total_budget ≥ sum budget required by *
}
```

The dual syntax for functionality is also available (Listings 26 and 27).



Listing 26: sumc3.mcdp

```

mcdp {
  provides total_power [W]

  sub g1 = instance `Generator1
  sub g2 = instance `Generator2
  sub g3 = instance `Generator3

  provided total_power ≤ (
    power provided by g1 +
    power provided by g2 +
    power provided by g3
  )
}

```

Listing 27: sumf3.mcdp

```

mcdp {
  provides total_power [W]

  sub g1 = instance `Generator1
  sub g2 = instance `Generator2
  sub g3 = instance `Generator3

  provided total_power ≤ ∑ power provided by *
}

```

## 4.5. Mathematical relations between functionalities and requirements

Once functionalities and resources are defined, it is possible to define relations between them by using various mathematical operations. The available math relations are shown in Table 4.1.

Table 4.1.: Math relations available

algebra	addition multiplication division (by a constant) subtraction (of a constant)
ceil/floor	ceil ceil0 floor floor0
exponentiation	a^2 sqrt pow
min/max	max min
approximation	approx

One thing to keep in mind is that these are not “functions”; rather, they are *relations*.

### 4.5.1. Abstract interpretation of mathematical relations

The following examples (Listings 28 and 29) show how the compiler is able to use *abstract interpretation* to infer the type of the result of the operations.

In Listing 28 we start with one requirement defined to be a natural number, and then we define several functionalities that are the result of applying different operations to that requirement. The compiler is able to infer the subset of numbers that are possible feasible values for each relation.

For example, for the line

```
provides f4 = r + 2.5
```

the compiler is able to infer that the smallest set that contains the optimal feasible functionalities for the relation is the set of numbers of the form  $r + 2.5$  where  $r$  is a natural number. This is written compactly in the figure as  $\geq 2.5 : 1$ .

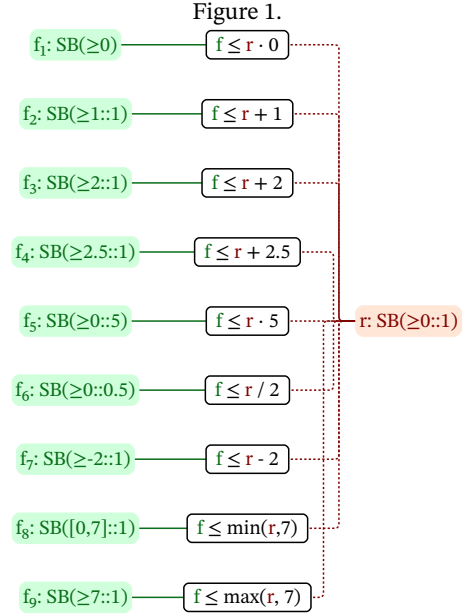
Likewise, for the line

```
provides f8 = min(r, 7)
```

the compiler is able to infer that the smallest set that contains the optimal feasible functionalities for the relation is the set of integers from 0 to 7, written compactly in the figure as  $[0, 7] : : 1$ .

Listing 28: numerics2.mcdp

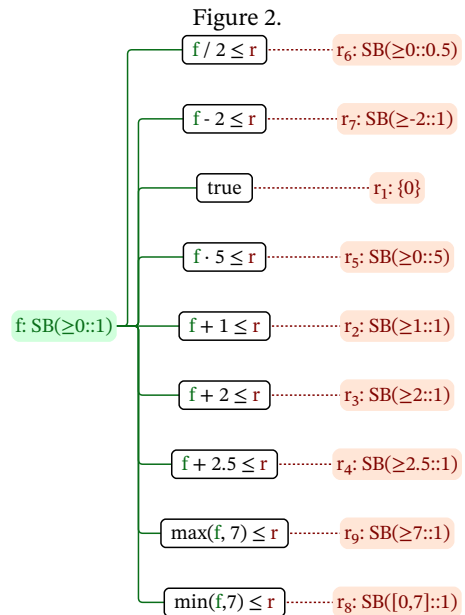
```
mcdp {
  requires r [N]
  provides f1 = required r · 0
  provides f2 = required r + 1
  provides f3 = required r + 2.0
  provides f4 = required r + 2.5
  provides f5 = required r · 5
  provides f6 = required r / 2
  provides f7 = required r - 2
  provides f8 = min(required r, 7)
  provides f9 = max(required r, 7)
}
```



The next example is symmetric, with the functionality being a natural number and the requirements being the result of applying different operations to that functionality.

Listing 29: numerics.mcdp

```
mcdp {
  provides f [N]
  requires r1 = provided f · 0
  requires r2 = provided f + 1
  requires r3 = provided f + 2.0
  requires r4 = provided f + 2.5
  requires r5 = provided f · 5
  requires r6 = provided f / 2
  requires r7 = provided f - 2
  requires r8 = min(provided f, 7)
  requires r9 = max(provided f, 7)
}
```



### 4.5.2. Min and max

In most cases, we can think of the `min` and `max` symbols as the usual mathematical functions. (Corresponding to the “meet” and “join” operations in lattice theory.)

However, in the context of MCDPs, the semantics is more subtle, and it holds even when the poset in question is not a lattice, so that the meet and join are not necessarily defined.

The following table shows the semantics of the `min` and `max` operations in MCDPs.

MCDPL expression	Equivalent semantics
<code>r ≥ min(a, b)</code>	$(r \geq a) \vee (r \geq b)$
<code>r ≥ max(a, b)</code>	$(r \geq a) \wedge (r \geq b)$
<code>f ≤ min(c, d)</code>	$(f \leq c) \wedge (f \leq d)$
<code>f ≤ max(c, d)</code>	$(f \leq c) \vee (f \leq d)$

If the poset is a lattice, then the semantics of the `min` and `max` operations are the same as the meet and join operations.

Consider the following poset with two elements:

Listing 30: `discrete.mcdp_poset`

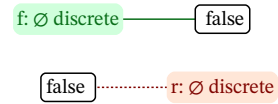
```
poset {
  a b
}
```

Because it is a discrete poset, the meet and join of the two elements do not exist. So,  $\min(a, b)$  and  $\max(a, b)$  are not defined if we consider them as mathematical functions. However, using the relation semantics we can write the following MCDP. We can see in the figure that the relation is reduced to the false relation.

Listing 31: `exampleminmax1.mcdp`

```
mcdp {
  provides f [discrete]
  requires r [discrete]
  provided f ≤ min(discrete: a, discrete: b)
  required r ≥ max(discrete: a, discrete: b)
}
```

Figure 3.



### 4.5.3. Floor and ceil relations

The `floor` and `ceil` operations are used to round down and up, respectively.

The `floor0` and `ceil0` are variants, defined as follows:

$$\text{floor}_0(x) = \begin{cases} 0 & \text{for } x = 0 \\ \text{ceil}(x - 1) & \text{for } x > 0 \end{cases} \quad (1)$$

$$\text{ceil}_0(x) = \begin{cases} 0 & \text{for } x = 0 \\ \text{floor}(x + 1) & \text{for } x > 0 \end{cases} \quad (2)$$

The functions `floor` and `floor0` agree everywhere except at nonzero integers. For example,  $\text{floor}(2) = 2$  and  $\text{floor}_0(2) = 1$ . Likewise, `ceil` and `ceil0` agree everywhere, except at nonzero integers:  $\text{ceil}(2) = 2$  and  $\text{ceil}_0(2) = 3$ .

The reason we need these extra operations is that relations need to be upper-semicontinuous if acting on requirements, and lower-semicontinuous if acting on functionalities.

In fact, we know that `floor` is upper semi-continuous:

$$\lim_{x \rightarrow 3^+} \text{floor}(x) = 3 \quad (3)$$

but that it is not lower semi-continuous:

$$\lim_{x \rightarrow 3^-} \text{floor}(x) = \text{does not exist} \quad (4)$$

The variant  $\text{floor}_0$  is lower semi-continuous:

$$\lim_{x \rightarrow 3^-} \text{floor}_0(x) = 3 \quad (5)$$

but it is not upper semi-continuous:

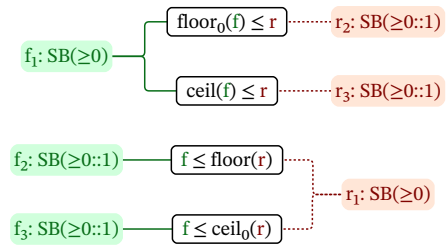
$$\lim_{x \rightarrow 3^+} \text{floor}_0(x) = \text{does not exist} \quad (6)$$

The following example shows the use of these operations.

Listing 32: numerics3.mcdp

```
mcdp {
  requires r1 [dimensionless]
  provides f1 [dimensionless]
  provides f2 = floor(required r1)
  provides f3 = ceil0(required r1)
  requires r2 = floor0(provided f1)
  requires r3 = ceil(provided f1)
}
```

Figure 4.



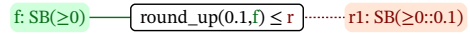
#### 4.5.4. Built-in approximations

The `approx` keyword is used to “discretize” a signal to a certain resolution. In practice, this means rounding the quantity to the nearest multiple of the resolution. The approximations must be conservative, so the functionalities are rounded down, and the requirements are rounded up. The following examples shows the symmetry between the functionalities and the requirements.

Listing 33: approx1.mcdp

```
mcdp {
  provides f [dimensionless]
  requires r1 = approx(provided f, 0.1)
}
```

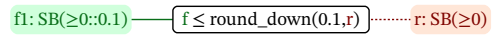
Figure 5.



Listing 34: approx2.mcdp

```
mcdp {
  requires r [dimensionless]
  provides f1 = approx(required r, 0.1)
}
```

Figure 6.



## 4.6. Accessing the components of a product

We have seen in Section 3.5 how to define anonymous and named products.

### 4.6.1. Accessing the components of an anonymous product

The “`take`” operation allows us to access the elements of a product. The syntax is:

```
take(signal, index)
```

For example, writing

```
mcdp {
  provides out [J x A]

  take(provided out, 0) ≤ 10 J
  take(provided out, 1) ≤ 2 A
}
```

is equivalent to writing

```
mcdp {
  provides out [J x A]
  provided out ≤ (10 J, 2 A)
}
```

### Accessing elements of a named product by name

If the product is a named product, it is possible to index those entries using one of these two syntaxes:

```
take(requirement, label)
```

```
take(functionality, label)
```

There is also a syntax with the dot, reminiscent of the syntax used in object-oriented languages:

```
(requirement).label
```

```
(functionality).label
```

For example:

```
mcdp {
  provides out [product(energy: J, current: A)]

  (provided out).energy ≤ 10 J
  (provided out).current ≤ 2 A
}
```

## 4.7. Operations on NDPS

The language includes a set of operations to manipulate NDPS. These are rarely used directly, but they are useful to understand the internal processing of NDPS.

Abstraction	<code>abstract NDP</code>
Compactification	<code>compact NDP</code>
Flattening	<code>flatten NDP</code>
Canonical form	<code>canonical NDP</code>

### 4.7.1. `compact`: Compactification

The construct `compact` takes an NDP and produces another in which parallel edges are compacted into one edge. This is one of the steps required for the solution of the MCDP.

The syntax is:

```
compact NDP
```

For every pair of NDPS that have more than one edge between them, those edges are being replaced.

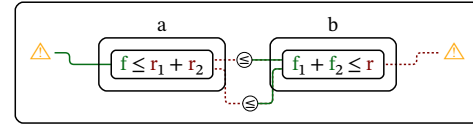
For example, consider this MCDP:

Listing 35: compact\_example\_1.mcdp

```
mcdp {
  sub a = instance mcdp {
    provides f [N]
    requires r1 [N]
    requires r2 [N]

    provided f ≤ (required r1 + required r2)
  }
  sub b = instance mcdp {
    provides f1 [N]
    provides f2 [N]
    requires r [N]

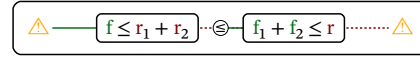
    required r ≥ (provided f1 + provided f2)
  }
  r1 required by a ≤ f1 provided by b
  r2 required by a ≤ f2 provided by b
}
```



The compacted version has only one edge between the two NDPs, with the corresponding poset being the product of the two posets:

Listing 36: compact\_example\_2.mcdp

```
compact `compact_example_1
```

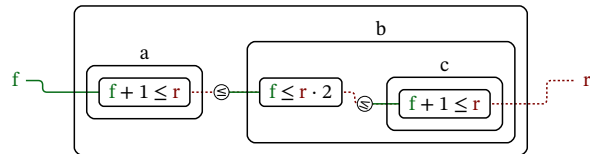


#### 4.7.2. flatten: Flattening

It is easy to create recursive composition in MCDPL, in which we have NDPs that contain other NDPs.

Listing 37: Composition1.mcdp

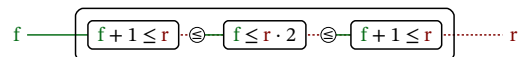
```
mcdp {
  T = mcdp {
    provides f [N]
    requires r [N]
    provided f + 1 ≤ required r
  }
  sub a = instance T
  sub b = instance mcdp {
    provides f [N]
    requires r [N]
    sub c = instance T
    provided f ≤ 2 * f provided by c
    required r ≥ r required by c
  }
  r required by a ≤ f provided by b
  requires r for b
  provides f using a
}
```



The “flattening” operation erases the borders between subproblems.

Listing 38: Composition1\_flattened.mcdp

```
flatten `Composition1
```



### 4.7.3. `abstract`: Abstraction

The command `abstract` takes an NDP and creates another NDP that forgets the internal structure.

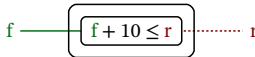
The syntax is:

```
abstract NDP
```

The resulting NDP is guaranteed to be equivalent to the initial one, but the internal structure is “forgotten”.

Listing 39: `m0.mcdp`

```
mcdp {
  provides f [m]
  requires r [m]
  required r ≥ provided f + 10 m
}
```



Listing 40: `m0_abstracted.mcdp`

```
abstract `m0
```



### 4.7.4. `canonical`: Canonical form

This puts the MCDP in a canonical form:

```
canonical NDP
```

The canonical form is obtained by compacting all the loops into one single loop.

## 4.8. `choose`: Union of design problems

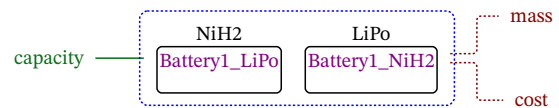
The `choose` construct allows describing the idea of “alternatives”.

The syntax is as follows:

Listing 41: `Batteries2.mcdp`

```
choose(
  NiH2: `Battery1_LiPo,
  LiPo: `Battery1_NiH2
)
```

Figure 7.



For a full example, see Section 1.6.

## 5. Higher-order modeling

### 5.1. Interfaces

A template has parameters that are constrained to have a certain *interface*.

In MCDPL, interfaces are first-class citizens. They reside in `.mcdp_interface` files, and can be imported and used in other libraries just like the other top-level entities.

Interfaces are defined using the `interface` keyword.

#### 5.1.1. `interface`: Declaring interfaces

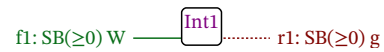
The syntax for declaring an interface is a subset of the syntax for declaring a MCDP. Only the definitions of functionality and resources are allowed.

For example, this interface declares one functionality and one resource:

Listing 42: `Int1.mcdp_interface`

```
interface {
  provides f1 [W]
  requires r1 [g]
}
```

Figure 1.



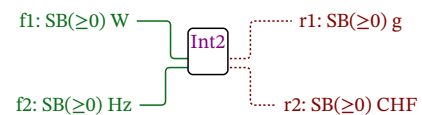
#### 5.1.2. `extends`: Extending interfaces

It is possible to *extend* an interface by adding more functionality or resources. This is done using the `extends` keyword.

Listing 43: `Int2.mcdp_interface`

```
interface {
  extends `Int1
  provides f2 [Hz]
  requires r2 [CHF]
}
```

Figure 2.



#### 5.1.3. `implements`: Using interfaces when defining models

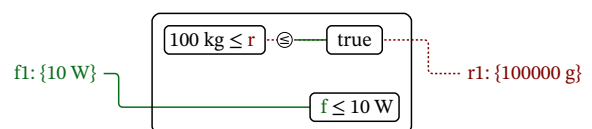
When defining a model, you can specify that it *implements* an interface. This means that the model must provide the functionality and resources required by the interface.

This is done using the `implements` keyword:

Listing 44: `Impl1.mcdp`

```
mcdp {
  implements `Int1
  provided f1 ≤ 10 W
  required r1 ≥ 100 kg
}
```

Figure 3.





## 5.2. Templates

Templates are contained in files with extension `.mcdp_template`.

### 5.2.1. `template`: Declaring templates

The syntax uses the keyword “`template`”. It is followed by square brackets, which specify the names of the interfaces for the template holes.

```
template [name1: interface1, name2: interface2]
mcdp {
    # usual definition here
}
```

For example, we can define the following simple interface:

Listing 45: `Scalar.mcdp_interface`

```
interface {
    provides f [dimensionless]
    requires r [dimensionless]
}
```

Figure 4.

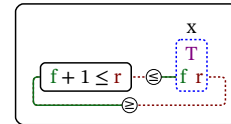


And then we can define a template that uses this interface:

Listing 46: `Loop1.mcdp_template`

```
template [T: `Scalar]
mcdp {
    sub x = instance T
    f provided by x ≥ r required by x + 1
}
```

Figure 5.

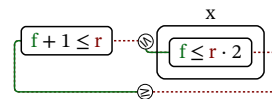


### 5.2.2. `specialize`: Instantiating templates

To instantiate a template, we need to specify which models to plug in it its holes. We use the keyword “`specialize`” as follows:

Listing 47: `Loop1Instance.mcdp`

```
specialize [
    T: mcdp {
        implements `Scalar
        provided f ≤ 2 · required r
    }
] `Loop1
```



## 6. Queries

### 6.1. Defining queries

*Queries* are specified in files with extension `.mcdp_query`. This file contains a YAML-document that specifies:

- the model that the query is about;
- the type of query;
- the parameters of the query;

There are two types of queries: `FixFunMinRes` and `FixResMaxFun`.

#### 6.1.1. FixFunMinRes

This query type is used to find the minimum amount of requirements required to achieve a certain functionality.

Consider a model `query1_model` that has 2 functionalities and 2 requirements:

```
mcdp {  
  provides f1 [m]  
  provides f2 [m]  
  
  requires r1 [m]  
  requires r2 [m]  
  
  provided f1 ≤ floor(r1)  
  r1 ≥ 1 m  
  r2 ≥ f2 + 1 m  
}
```

A query that uses this model could look like this:

```
title: Query 1  
description: ''  
model: ``query1_model``  
query:  
  query_type: FixFunMinRes  
  min_f:  
    f_1: '1 m'  
    f_2: '2 m'  
  max_r:  
    r_1: '3 m'  
    r_2: '4 m'  
  optimize_for: [r_1]
```

The fields `title` and `description` are just used for metadata.

The `model` field should be a string that evaluates to a model.

The `query` field specifies the details of the query:

- The field `type` should be set to `FixFunMinRes`.
- The field `min_f` is a dictionary that specifies the minimum values for the functionalities.
- The field `max_r` is a dictionary that specifies the upper bound for the requirements.

- The field `optimize_for` is a list that specifies the resources to optimize for.

In this case, the query is equivalent to this optimization problem:

$$\begin{aligned} \min & & r_1 & & (1) \\ \text{such that} & & \text{query1\_model}(\langle 1\text{m}, 2\text{m} \rangle, \langle r_1, r_2 \rangle) \text{ is feasible} & & (2) \\ & & r_1 \leq 3\text{m} & & (3) \\ & & r_2 \leq 4\text{m} & & (4) \end{aligned}$$

We are minimizing the value of `r_1`, which is the only element in the `optimize_for` list. The value of `r_2` is not relevant, although it is still bounded.

### 6.1.2. FixResMaxFun

This query type is used to find the maximum amount of functionalities that can be achieved with a certain amount of resources.

The syntax is similar to the previous type of query:

```
title: Query 2
description: ''
model: ``query1_model``
query:
  query_type: FixResMaxFun
  min_f:
    f_1: '1 m'
    f_2: '2 m'
  max_r:
    r_1: '3 m'
    r_2: '4 m'
  optimize_for: [f_1]
```

The query is equivalent to this optimization problem:

$$\begin{aligned} \max & & f_1 & & (5) \\ \text{such that} & & \text{query1\_model}(\langle f_1, f_2 \rangle, \langle 3\text{m}, 4\text{m} \rangle) \text{ is feasible} & & (6) \\ & & f_1 \geq 1\text{m} & & (7) \\ & & f_2 \geq 2\text{m} & & (8) \end{aligned}$$

## 7. Syntax

### 7.1. MCDPL syntax

#### 7.1.1. Characters

A MCDP file is a sequence of Unicode code-points that belong to one of the classes described in Table 7.1. All files are assumed to be encoded in UTF-8.

Table 7.1.: Character classes

class	characters
Latin letters	abcdefghijklmnopqrstuvwxyz ABCDEFGHIJKLMNOPQRSTUVWXYZ
Underscore	_
Greek letters	$\alpha\beta\gamma\delta\epsilon\zeta\eta\theta\iota\kappa\lambda\mu\nu\xi\pi\rho\sigma\tau\upsilon\phi\chi\psi\omega$ $\Gamma\Delta\Theta\Lambda\Xi\P\Sigma\Upsilon\Phi\Psi\Omega$
Digits	0123456789
Superscripts	$x^1\ x^2\ x^3\ x^4\ x^5\ x^6\ x^7\ x^8\ x^9$
Subscripts	$x_1\ x_2\ x_3\ x_4\ x_5\ x_6\ x_7\ x_8\ x_9$
Comment delimiter	#
String delimiters	'"
Backtick	`
Parentheses	[]{}()
Operators	<= >= <≤, >≥
Tuple-making	< > ⟨ ⟩
Arrows glyphs	<--   <-->   -->    ↔ ⇔ ⇌
Math	= . * - + ^
Other glyphs	$\times \top \perp \mathcal{P} \mathbb{N} \mathbb{R} \mathbb{Z} \sum \pm \uparrow \downarrow \infty \emptyset \$$

#### 7.1.2. Comments

Comments work as in Python. Anything between the symbol # and a newline is ignored. Comments can include any Unicode character.

#### 7.1.3. Reserved keywords

The reserved keywords are shown in Table 7.2.

#### 7.1.4. Syntactic equivalence

MCDPL allows a number of Unicode glyphs as abbreviations of a few operators.

For example, every occurrence of a superscript of the digit  $d$  is interpreted as a power  $^d$ . It is syntactically equivalent to write  $x^2$  or  $x^2$ .

Other syntactic equivalences are shown in Table 7.3.

#### 7.1.5. Identifiers

An *identifier* is a string that is not a reserved keyword and follows these rules:

1. It starts with a Latin or Greek letter (except underscore).
2. It contains Latin letters, Greek letters, underscore, digit,
3. It ends with Latin letters, Greek letters, underscore, digit, or a subscript.

Table 7.2.: Reserved keywords

abstract	by	implements	required
add_bottom	canonical	instance	requires
and	catalog	Int	Reals
approx_lower	choose	interface	solve_f
approx_upper	code	lowerclosure	solve_r
approx	compact	lowersets	solve
approxu	constant	maximals	specialize
assert_empty	coproduct	mcdp	sum
assert_equal	emptyset	minimals	take
assert_geq	eversion	namedproduct	template
assert_gt	extends	Nat	top
assert_leq	flatten	poset	uncertain
assert_lt	for	powerset	upperclosure
assert_nonempty	ignore_resources	product	uppersets
between	ignore	provided	using
bottom	implemented-by	provides	variable

Table 7.3.: Unicode glyphs and syntactically equivalent ASCII

Unicode	ASCII
$\leq$ or $\preceq$	<code>&lt;=</code>
$\geq$ or $\succeq$	<code>&gt;=</code>
$\cdot$	<code>*</code>
$\langle \rangle$	<code>&lt; &gt;</code>
$\top$	<code>Top</code>
$\perp$	<code>Bottom</code>
$\mathcal{P}$	<code>powerset</code>
$\pm$	<code>+-</code>
$\sum$	<code>sum</code>
$\mapsto$	<code> --&gt;</code>
$\leftarrow$	<code>&lt;-- </code>
$\leftrightarrow$	<code>&lt;--&gt;</code>
$\emptyset$	<code>Emptyset</code>
$\mathbb{N}$	<code>Nat</code>
$\mathbb{R}$	<code>Rcomp</code>
$\mathbb{Z}$	<code>Int</code>
$\uparrow$	<code>upperclosure</code>
$\downarrow$	<code>lowerclosure</code>
$\times$	<code>x</code>

A regular expression that captures these rules is:

```
identifier = [latin|greek][latin|greek|_|digit]*[latin|greek|_|digit|subscript]?
```

Here are some examples of good identifiers: `a`, `a_4`, `a4`, `alpha`,  `$\alpha$` .

### 7.1.6. Use of Greek letters as part of identifiers

MCDPL allows to use some Unicode characters, Greek letters and subscripts, also in identifiers and expressions. For example, it is equivalent to write `alpha_1` and  `$\alpha_1$` .

For subscripts, every occurrence of a subscript of the digit  $d$  is converted to the fragment `_d`.

Subscripts can only occur at the end of an identifier: `a_1` is valid, while `a_1b` is not a valid identifier.




Every Greek letter is converted to its name. It is syntactically equivalent to write `alpha_material` or  `$\alpha$ _material`.

Greek letter names are only considered at the beginning of the identifier and only if they are followed by a non-word character. For example, the identifier `alphabet` is not converted to  `$\alpha$ bet`.

Part C.

Software manual

## 8. MCDP Command line interface






`mcdp` is a command-line interface (CLI) tool available for  Linux,  macOS, and  Windows.

The binaries are available from the page

<https://github.com/zupermind/releases/releases/latest>

The table below summarizes the platform support.

Table 8.1.: Platform support matrix for MCDP CLI

Operating System	Architecture	Status
 Ubuntu 22	x86_64, ARM64	Supported
 Ubuntu 24	x86_64, ARM64	Supported
 Debian	x86_64, ARM64	Supported
 macOS 15	Intel, Apple Silicon	Supported
 Windows 11	x86_64, ARM64	Experimental

### 8.1. Installation

#### 8.1.1. Prerequisites

##### Docker

A working Docker installation is required

- [Install Docker Desktop](#) ( Windows/ macOS)
- [Install Docker Engine](#) ( Linux)

#### 8.1.2. Linux (Ubuntu/Debian)

Recent versions of Ubuntu and Debian are supported directly.

The executables are likely to work also on other Linux distributions, but they are not tested.

Installation is straightforward:

1. Download the binary for your architecture
2. Make it executable:

```
| chmod +x mcdp-cli-*
```

3. (Optional) Move to system path:


```
| sudo mv mcdp-cli-* /usr/local/bin/mcdp
```

4. Verify installation:

```
| mcdp version
```

#### 8.1.3. macOS

The installation process is similar to Linux.

** About security notices:** If you download the executables using a browser, the executable will be marked as untrusted and quarantined. Use the command line to avoid security warnings.

## Recommended Installation (via Terminal)

```
# For Apple Silicon Macs (M1/M2/M3)
curl -L -o mcdp https://github.com/zupermind/mcdp-binaries/releases/latest/download/mcdp-[VERSION]-macos15-arm64
```

```
# For Intel Macs
curl -L -o mcdp https://github.com/zupermind/mcdp-binaries/releases/latest/download/mcdp-[VERSION]-macos15-amd64
```

```
# Make it executable
chmod +x mcdp

# Verify installation
./mcdp version

# (Optional) Move to system path
sudo mv mcdp /usr/local/bin/
```

Replace [VERSION] with the actual version number from the releases page.

### 8.1.4. 🖥️ Windows (Experimental)

**⚠️ Note:** Windows support is currently experimental. Please report any issues you encounter.

#### Option 1: Windows Installer (Recommended)

Download and run the installer for your architecture:

- **x64:** mcdp-[VERSION]-windows-amd64-installer.exe
- **ARM64:** mcdp-[VERSION]-windows-arm64-installer.exe

The installer will:

- Install the MCDP CLI to C:\Program Files\MCDP
- Optionally add it to your system PATH (recommended)
- Allow uninstallation through Windows Settings

**❗ Important:** After installation, you'll need to open a new PowerShell or Command Prompt window for the PATH changes to take effect.

#### Option 2: Manual Installation (PowerShell/Command Prompt)

1. Download the standalone .exe file for your architecture
2. **If downloaded via browser:** You may see security warnings – click “More info” → “Run anyway”

#### Manual Installation via PowerShell

```
# For x64
curl -L -o mcdp.exe https://github.com/zupermind/mcdp-binaries/releases/latest/download/mcdp-[VERSION]-windows-amd64.exe

# For ARM64
curl -L -o mcdp.exe https://github.com/zupermind/mcdp-binaries/releases/latest/download/mcdp-[VERSION]-windows-arm64.exe

# Verify installation
.\mcdp.exe version
```

**Tip:** This is a CLI tool. If you double-click the .exe file, you'll see nothing; you have to use it from a terminal.



### 8.1.5. Getting Started

Once installed, you can:

```
# View help and available commands
mcdp help

# Check version
mcdp version
```

**Note:** On Windows, the executable is `mcdp.exe`, but you can simply type `mcdp` in your terminal.

### 8.1.6. Troubleshooting

#### “Command not found” error

- Ensure the binary is in your system PATH or use the full path to the executable

#### Permission denied (🐧 Linux/🍏 macOS)

- Run `chmod +x mcdp` to make the file executable

#### Security warnings (🪟 Windows/🍏 macOS)

- Use the command-line installation method to avoid browser quarantine
- On Windows, you may need to add an exception in Windows Defender

#### Docker not found

- Ensure Docker is installed and running
- On Linux, you may need to add your user to the docker group: `sudo usermod -aG docker $USER`

## 8.2. Command `mcdp update` - Self updating support

The MCDP CLI includes a built-in self-update feature:

```
mcdp update
```

This command will:

- Check for the latest version
- Download and install it automatically
- Preserve your current settings

**🪟 Note for Windows users** You may need to run PowerShell as Administrator when using the `mcdp update` command if MCDP was installed with the installer to `C:\textbackslash Program Files\textbackslash MCDP`.

## 8.3. Command `mcdp co-design plot`

`mcdp co-design plot` is a command that can be used to draw produce various visualizations of MCDP models and posets. It can be used to generate the various visualizations used in this book.

The basic calling syntax is:

```
mcdp co-design plot --plots PLOT_NAMES THING_NAME
```

where `PLOT_NAMES` are comma-separated plot names, and `THING_NAME` is the name of the MCDP problem to plot.

To see the complete set of plot names, look at the output of

```
| mcdp co-design plot --help
```

For example, consider the following model:

Listing 48: ExampleModel.mcdp

```
mcdp {
  provides capacity [J]
  requires mass [g]
  ρ = 100 kWh / kg # specific_energy
  required mass ≥ provided capacity / ρ
}
```

Here are some examples of plots that can be generated by the command.

```
| mcdp co-design plot --plots ndp_gojs_nowrap_noexpand ExampleModel
```

capacity —  $f / 100 \leq r$   $\oplus$   $f [J*kg/kWh] \leq r [g]$  ..... mass

```
| mcdp co-design plot --plots ndp_gojs_wrap_noexpand_units ExampleModel
```

capacity: SB( $\geq 0$ ) J —  $f / 100 \leq r$   $\oplus$   $f [J*kg/kWh] \leq r [g]$  ..... mass: SB( $\geq 0$ ) g

```
| mcdp co-design plot --plots ndp_gojs_interface ExampleModel
```

capacity: SB( $\geq 0$ ) J — ExampleModel ..... mass: SB( $\geq 0$ ) g

## 8.4. Command `mcdp co-design solve` - Solving MCDP problems (legacy)

The command `mcdp co-design solve` is used to solve `FixFunMinReq` queries. It is useful as a quick way to test the models. The command `mcdp co-design solve-query` offers more functionality.

The basic calling syntax is:

```
| mcdp co-design solve MODEL PARAMETERS
```

The parameter `MODEL` is the name of the model to solve.

The parameter `PARAMETERS` is the parameters to solve the query.

A typical invocation is:

```
| mcdp co-design solve model_name "600 J"
```

The full usage is:

```
Usage: mcdp co-design solve [OPTIONS] MODEL PARAMETERS
```

Options:

-C <PATH> Run as if the command was started in the given directory

-d <DIRECTORY> Source directory [default: .]

-o <OUTDIR> [default: out-mcdp-solve]

-v, --verbose...

Enable verbose output

--nocache Do not use cache

--imp Compute and show implementations

--show-model Show the resulting NDP

--show-dp Show the resulting DP

--pessimistic <PESSIMISTIC> Resolution for pessimistic solution

--optimistic <OPTIMISTIC> Resolution for optimistic solution

-h, --help Print help

### 8.4.1. Resolution options

The options `--optimistic` and `--pessimistic` can be used to specify the resolution for the solution.

```
| mcdp co-design solve model_name --optimistic 10 --pessimistic 10 "600 J"
```

### 8.4.2. Implementation options

The option `--imp` can be used to compute the implemnetations.

```
| mcdp co-design solve model_name --imp "600 J"
```

## 8.5. Command `mcdp co-design solve-query` - Solving queries on MCDP problems

The command `mcdp co-design solve-query` is used to solve queries specified as query files.

The format for query files is documented in Section 6.1.

Usage: `mcdp co-design solve-query [OPTIONS] <QUERY>`

Arguments:

`<QUERY>` Query to solve

Options:

<code>-C &lt;PATH&gt;</code>	Run as if the command was started in the given directory
<code>-d &lt;DIRECTORY&gt;</code>	Source directory [default: .]
<code>-o &lt;OUTDIR&gt;</code>	Source directory [default: out-solve-query]
<code>-v, --verbose...</code>	Enable verbose output
<code>--nocache</code>	Do not use cache
<code>--imp</code>	Compute and show implementations
<code>--blueprints</code>	Compute and show blueprints
<code>--pessimistic &lt;PESSIMISTIC&gt;</code>	Resolution for pessimistic solution
<code>--optimistic &lt;OPTIMISTIC&gt;</code>	Resolution for optimistic solution
<code>--imp-recheck</code>	Activates paranoid mode, rechecking implementation if <code>--imp</code> was provided
<code>--blueprints-recheck</code>	Activates paranoid mode, rechecking blueprints if <code>--blueprints</code> was provided
<code>-h, --help</code>	Print help

The basic calling syntax is:

```
| mcdp co-design solve-query QUERY_NAME
```

### 8.5.1. Resolution options

The options `--optimistic` and `--pessimistic` can be used to specify the resolution for the solution.

```
| mcdp co-design solve-query --optimistic 10 --pessimistic 10 myquery
```

### 8.5.2. `--imp` - Computing implementations

The option `--imp` can be used to compute the implemnetations.

The option `--imp-recheck` can be used to recheck that the implementations are correct.

### 8.5.3. `--blueprints` - Computing blueprints

The option `--blueprints` can be used to compute the blueprints.

The option `--blueprints-recheck` can be used to recheck the blueprints.

## 8.6. Command `mcdp co-design export` - Exporting MCDP problems

The command `mcdp co-design export` allows exporting compiled MCDP problems to a standard YAML/CBOR format, which is described in Part F.

The calling syntax is:

```
Usage: mcdp co-design export [OPTIONS]

Options:
-C <PATH> Run as if the command was started in the given directory
-d <DIRECTORY> Source directory [default: .]
-o <OUTPUT> Destination directory for the exported data
-v, --verbose...
Enable verbose output
-f, --format <FORMAT> Export format [default: yaml] [possible values: yaml, cbor]
-h, --help Print help
```

For example, suppose that the directory `data` contains the following libraries:

```
data/
  lib1/
    model1.mcdp
    poset1.mcdp
```

Then the command will export the contents of the libraries to the directory `out`.

```
| mcdp co-design export -d data -o out
```

The directory `out` will contain the following files:

```
out/
  lib1/
    models/
      model1.ndp.mcdp2.yaml
      model1.dp.mcdp2.yaml
      model1.dpc.mcdp2.yaml
    posets/
      poset1.poset.mcdp2.yaml
```

All files end in `.mcdp2.yaml` to signify that they are in the MCDP format 2 and represented as YAML.

For each model, the command will export 3 files:

- `model1.ndp.mcdp2.yaml`: This contains the parsed model as a Named DP, a graph. See Section 26.13.
- `model1.dp.mcdp2.yaml`: This contains the DP of the model. See Section 26.12.
- `model1.dpc.mcdp2.yaml`: This contains the *compiled DP* of the model. See Section 26.12.28.

## 9. Libraries for parsing the exported data

This chapter describes the libraries that are available for parsing the data exported by the `mcdp co-design export` command (see Section 8.6).


The OpenAPI specification for the MCDP format 2 is available at on GitHub.

### 9.1. Python library `mcdp-format2-py`

The Python library `mcdp-format2-py` is a Python library for parsing the exported data.

It is available on PyPI and can be installed with:

```
pip install mcdp-format2-py
```

The source code is available on GitHub at  [zupermind/mcdp-format2-py](https://github.com/zupermind/mcdp-format2-py).

Once installed, the library can be used to parse the exported data.

```
from mcdp_format2_py import load
data = load("model1.ndp.mcdp2.yaml")
print(data)
```

There is a command line tool `mcdp-format2-py-load` that can be used to test the library.


```
mcdp-format2-py-load model1.ndp.mcdp2.yaml
```

### 9.2. Rust crate `mcdp-format2-rs`

The Rust crate `mcdp-format2-rs` is a Rust crate for parsing the exported data.

It is available on crates.io and can be installed with:

```
cargo add mcdp-format2-rs
```

The source code is available on GitHub at  [zupermind/mcdp-format2-rs](https://github.com/zupermind/mcdp-format2-rs).

There is a command line tool `mcdp-format2-rs-load` that can be used to test the crate.

```
cargo install mcdp-format2-rs
mcdp-format2-rs-load model1.ndp.mcdp2.yaml
```

## Part D.

### Mathematical underpinnings for computational co-design

## 10. Order theory

In this chapter we recall some basic notions of order theory and describe our notation.

### 10.1. Posets

#### Definition 10.1

A *pre-order*  $\mathbf{P}$  is a set  $\mathbf{P}$  equipped with a binary relation  $\leq_{\mathbf{P}}$  that is reflexive and transitive. A pre-order is a *poset* if the relation is also antisymmetric.

### 10.2. Special subsets of posets

#### Definition 10.2

An *antichain* in a poset  $\mathbf{P}$  is a subset  $\mathbf{A} \subseteq \mathbf{P}$  such that for all distinct elements  $x, y \in \mathbf{A}$ ,  $x \leq_{\mathbf{P}} y$  does not hold.

$\text{Anti } \mathbf{P}$  is the set of all antichains in  $\mathbf{P}$ .

#### Definition 10.3

A *lower set* in a poset  $\mathbf{P}$  is a subset  $\mathbf{S} \subseteq \mathbf{P}$  such that  $\forall x \in \mathbf{S}, \forall p \in \mathbf{P} : p \leq_{\mathbf{P}} x \Rightarrow p \in \mathbf{S}$ .

#### Definition 10.4

An *upper set* in a poset  $\mathbf{P}$  is a subset  $\mathbf{S} \subseteq \mathbf{P}$  such that  $\forall x \in \mathbf{S}, \forall p \in \mathbf{P} : x \leq_{\mathbf{P}} p \Rightarrow p \in \mathbf{S}$ .

#### Definition 10.5

Given a poset  $\mathbf{P}$ ,  $\text{Pow } \mathbf{P}$  is the poset of subsets of  $\mathbf{P}$ , ordered by inclusion.

#### Definition 10.6

Given a poset  $\mathbf{P}$ ,  $\text{UP}$  is the poset of upper sets in  $\mathbf{P}$ , ordered by inclusion.

#### Definition 10.7

Given a poset  $\mathbf{P}$ ,  $\text{LP}$ , is the poset of lower sets in  $\mathbf{P}$ , ordered by inclusion.

These constructions will also be indicated as  $\text{P\_C\_UpperSets}(\mathbf{P})$  and  $\text{P\_C\_LowerSets}(\mathbf{P})$ .

### 10.3. Monotone maps

#### Definition 10.8

A monotone map  $f : \mathbf{P} \rightarrow_{\text{pos}} \mathbf{Q}$  is a map  $f : \mathbf{P} \rightarrow \mathbf{Q}$  such that  $x \leq_{\mathbf{P}} y \Rightarrow f(x) \leq_{\mathbf{Q}} f(y)$ .

### 10.4. Closure operators

#### 10.4.1. Upper and lower closure of a point

**Definition 10.9** (Lower closure  $\downarrow$  in a poset)

For a poset  $\mathbf{P}$  we define the lower closure  $\downarrow_P$  as

$$\begin{aligned} \downarrow_P : \mathbf{P} &\rightarrow_{\text{Pos}} \mathbf{LP} \\ p &\longmapsto \{q \in \mathbf{P} \text{ such that } q \leq_P p\} \end{aligned} \quad (1)$$

Note that  $\downarrow$  is monotone: if  $p \leq_P q$  then  $\downarrow p \subseteq \downarrow q$ .

**Definition 10.10** (Upper closure  $\uparrow$  in a poset)  
For a poset  $\mathbf{P}$  we define the upper closure  $\uparrow_P$  as

$$\begin{aligned} \uparrow_P : \mathbf{P}^{\text{op}} &\rightarrow_{\text{Pos}} \mathbf{UP} \\ p^* &\longmapsto \{q \in \mathbf{P} \text{ such that } p^* \leq_P q\} \end{aligned} \quad (2)$$

Note that  $\uparrow$  is antitone (its domain is  $\mathbf{P}^{\text{op}}$ ): if  $p \leq_P q$  then  $\uparrow p \supseteq \uparrow q$ .

We also define the strict versions  $\downarrow$  and  $\uparrow$ :

**Definition 10.11** (Strict lower closure  $\downarrow$  in a poset)  
For a poset  $\mathbf{P}$  we define the strict lower closure  $\downarrow_P$  as

$$\begin{aligned} \downarrow_P : \mathbf{P} &\rightarrow_{\text{Pos}} \mathbf{LP} \\ p &\longmapsto \{q \in \mathbf{P} \text{ such that } q <_P p\} \end{aligned} \quad (3)$$

**Definition 10.12** (Strict upper closure  $\uparrow$  in a poset)  
For a poset  $\mathbf{P}$  we define the strict upper closure  $\uparrow_P$  as

$$\begin{aligned} \uparrow_P : \mathbf{P}^{\text{op}} &\rightarrow_{\text{Pos}} \mathbf{UP} \\ p^* &\longmapsto \{x \in \mathbf{P} \text{ such that } p^* <_P x\} \end{aligned} \quad (4)$$

#### 10.4.2. Upper and lower closure of a subset

Similarly, for a subset  $\mathbf{S} \subseteq \mathbf{P}$  we have that  $\uparrow \mathbf{S}$  is the upper closure of  $\mathbf{S}$  and  $\downarrow \mathbf{S}$  is the lower closure of  $\mathbf{S}$ .

**Definition 10.13** (Upper closure  $\uparrow$  of a subset in a poset)  
For a subset  $\mathbf{S} \subseteq \mathbf{P}$  we have that  $\uparrow \mathbf{S}$  is the upper closure of  $\mathbf{S}$ :

$$\begin{aligned} \uparrow_P : \text{Pow } \mathbf{P} &\rightarrow_{\text{Pos}} \mathbf{UP} \\ \mathbf{S} &\longmapsto \bigcup_{q \in \mathbf{S}} \uparrow q \end{aligned} \quad (5)$$

**Definition 10.14** (Lower closure  $\downarrow$  of a subset in a poset)  
For a subset  $\mathbf{S} \subseteq \mathbf{P}$  we have that  $\downarrow \mathbf{S}$  is the lower closure of  $\mathbf{S}$ :

$$\begin{aligned} \downarrow_P : \text{Pow } \mathbf{P} &\rightarrow_{\text{Pos}} \mathbf{LP} \\ \mathbf{S} &\longmapsto \bigcup_{q \in \mathbf{S}} \downarrow q \end{aligned} \quad (6)$$

Note that both  $\uparrow$  and  $\downarrow$  are monotone because the order is defined by inclusion.

#### 10.4.3. Upper and lower closure of a function

For a function  $f : \mathbf{P} \rightarrow \mathbf{Q}$  we define  $\uparrow f$  as the upper closure of the result of  $f$  and  $\downarrow f$  as the lower closure of the result of  $f$ .

**Definition 10.15** (Upper closure of a function)



Given a monotone map  $f : \mathbf{P} \rightarrow_{\mathbf{Pos}} \mathbf{Q}$ , we define the map  $\uparrow f$  as

$$\begin{aligned} \uparrow f : \mathbf{P}^{\text{op}} &\rightarrow_{\mathbf{Pos}} \mathbf{UQ} \\ p^* &\longmapsto \uparrow f(p^*) \end{aligned} \tag{7}$$

Note that  $\uparrow f$  is antitone. We define the strict version  $\uparrow\!\!\uparrow f$  analogously.

**Definition 10.16** (Lower closure of a function)

Given a monotone map  $g : \mathbf{Q} \rightarrow_{\mathbf{Pos}} \mathbf{P}$ , we define the map  $\downarrow g$  as

$$\begin{aligned} \downarrow g : \mathbf{Q} &\rightarrow_{\mathbf{Pos}} \mathbf{LP} \\ q &\longmapsto \downarrow g(q) \end{aligned} \tag{8}$$

Note that  $\downarrow g$  is monotone. We define the strict version  $\downarrow\!\!\downarrow g$  analogously.

**Lemma 10.17** (Composition of closure operators).

$$\uparrow(f \circ g) = \uparrow f \circ \uparrow g \tag{9}$$

$$\downarrow(f \circ g) = \downarrow f \circ \downarrow g \tag{10}$$

## 11. Design problems (DPs)

### 11.1. Design problems

**Definition 11.1**

Given two posets  $\mathbf{F}$  and  $\mathbf{R}$ , a *design problem* (DP)

$$\mathbf{d} : \mathbf{F} \rightarrow_{\mathbf{DP}} \mathbf{R} \quad (1)$$

is a monotone map

$$\mathbf{d} : \mathbf{F}^{\text{op}} \times \mathbf{R} \rightarrow_{\mathbf{Pos}} \mathbf{Bool} \quad (2)$$

**Lemma 11.2.**  $\mathbf{DP}$  is a traced monoidal category [1].

**Lemma 11.3.**  $\mathbf{DP}$  is a locally posetal category [1].

**Lemma 11.4.** Fixed two posets  $\mathbf{F}$  and  $\mathbf{R}$ , the homset  $\mathbf{DP}(\mathbf{F}, \mathbf{R})$  is a complete lattice.

### 11.2. $\mathbf{PosL}$ and $\mathbf{PosU}$

The two categories  $\mathbf{PosL}$  and  $\mathbf{PosU}$  are used to represent the query solutions for a DP.

**Definition 11.5 ( $\mathbf{PosL}$ )**

Given two posets  $\mathbf{kdom}$ ,  $\mathbf{kcod}$  a morphism of  $\mathbf{PosL}$

$$\ell : \mathbf{kdom} \rightarrow_{\mathbf{PosL}} \mathbf{kcod} \quad (3)$$

is a monotone map

$$\ell : \mathbf{kdom} \rightarrow_{\mathbf{Pos}} \mathbf{Lkcod} \quad (4)$$

*This construction is described by the schema  $\mathbf{L1Map}$  (Section 26.4).*

**Definition 11.6 ( $\mathbf{PosU}$ )**

Given two posets  $\mathbf{kdom}$ ,  $\mathbf{kcod}$  a morphism of  $\mathbf{PosU}$

$$u : \mathbf{kdom} \rightarrow_{\mathbf{PosU}} \mathbf{kcod} \quad (5)$$

is a monotone map

$$u : \mathbf{kdom}^{\text{op}} \rightarrow_{\mathbf{Pos}} \mathbf{Ukcod} \quad (6)$$

*This construction is described by the schema  $\mathbf{U1Map}$  (Section 26.5).*

Note that the domain is  $\mathbf{kdom}^{\text{op}}$ : as  $f$  increases, the solution set decreases  $u(f)$  decreases.

**Lemma 11.7.**  $\mathbf{PosU}$  and  $\mathbf{PosL}$  are traced monoidal categories that are locally posetal [1].

**Remark 11.8.** The book [1] uses a slightly different definition, by identifying a morphism of  $\mathbf{PosU}$  as a monotone map

$$\mathbf{kdom} \rightarrow_{\mathbf{Pos}} \mathbf{Ukcod} \quad (7)$$

with the domain being  $\mathbf{kdom}$  instead of  $\mathbf{kdom}^{\text{op}}$ . All results are still valid with this other convention.

### 11.3. Queries for DPs

**Definition 11.9** (DP queries)

Given a DP  $\mathbf{d} : \mathbf{F} \rightarrow_{\mathbf{DP}} \mathbf{R}$ , we can define the following *queries*:

$$\begin{aligned} \mathbf{FR}(\mathbf{d}) : \mathbf{F} &\rightarrow_{\mathbf{PosU}} \mathbf{R} \\ f^* &\longmapsto \{r \text{ such that } \mathbf{d}(f^*, r)\} \end{aligned} \tag{8}$$

$$\begin{aligned} \mathbf{RF}(\mathbf{d}) : \mathbf{R} &\rightarrow_{\mathbf{PosL}} \mathbf{F} \\ r &\longmapsto \{f \text{ such that } \mathbf{d}(f^*, r)\} \end{aligned} \tag{9}$$

**Lemma 11.10.**  $\mathbf{FR}$  and  $\mathbf{RF}$  can be seen as functors from  $\mathbf{DP}$  to  $\mathbf{PosU}$  and  $\mathbf{PosL}$ , respectively.

- $\mathbf{FR}$  is a functor  $\mathbf{DP} \rightarrow \mathbf{PosU}$
- $\mathbf{RF}$  is a contravariant functor  $\mathbf{DP}^{\text{op}} \rightarrow \mathbf{PosL}$
- $\mathbf{FR}$  and  $\mathbf{RF}$  preserve the traced monoidal structure.
- $\mathbf{FR}$  and  $\mathbf{RF}$  are monotone functors (they respect the posetal structure)

See [1] for proofs.

## 12. DP computability and well foundedness

This chapter discusses some computability concerns for DPs. In particular, we are interested in understanding when the upper/lower sets induced by the **FR** and **RF** operations can be described by (finite) antichains.

### 12.1. Well-foundedness

#### 12.1.1. Well-foundedness of upper and lower sets

**Definition 12.1** (Below well founded – BWF)

An upper set  $S \in \mathbf{UP}$  is *below well founded* (BWF) if there exists an antichain  $A \in \mathbf{Anti P}$  such that  $S = \uparrow A$ .

It is *finitely below well founded* (fBWF) if the supporting set is finite.

**Definition 12.2** (Above well founded – AWF)

A lower set  $S \in \mathbf{LP}$  is *above well founded* (AWF) if there exists an antichain  $A \in \mathbf{Anti P}$  such that  $S = \downarrow A$ .

We say that it is *finitely above well founded* (fAWF) if the supporting set is finite.

We call  $\mathbf{L}_w \mathbf{P}$  the set of all well-founded lower sets in  $\mathbf{P}$  and  $\mathbf{L}_f \mathbf{P}$  those that are finitely well-founded. Analogously we define  $\mathbf{U}_w \mathbf{P}$  and  $\mathbf{U}_f \mathbf{P}$ .

It is useful to interpret these sets as fixpoints of certain operators. For example, an antichain is a subset such that  $A = \text{Min } A$ , or, equivalently, such that  $A = \text{Max } A$ . So we can define the set of antichains as a fixpoint of the Min and Max operators:

$$\mathbf{Anti P} = \text{Fix}(\text{Min}_P) \quad (1)$$

$$= \text{Fix}(\text{Max}_P) \quad (2)$$

Likewise, we can define the lower sets as the fixpoint of the  $\downarrow$  operator, and the upper sets as the fixpoint of the  $\uparrow$  operator:

$$\mathbf{LP} = \text{Fix}(\downarrow_P) \quad (3)$$

$$\mathbf{UP} = \text{Fix}(\uparrow_P) \quad (4)$$

The well-founded upper and lower sets can be defined as these fixpoints:

$$\mathbf{U}_w \mathbf{P} = \text{Fix}(\text{Min} \circ \uparrow_P) \quad (5)$$

$$\mathbf{L}_w \mathbf{P} = \text{Fix}(\text{Max} \circ \downarrow_P) \quad (6)$$

#### 12.1.2. Well-foundedness of $\mathbf{PosU}$ and $\mathbf{PosL}$ morphisms

We can then propagate these properties to functions that map into upper and lower sets.

**Definition 12.3** (Below well founded – BWF)

We call a morphism  $u : \mathbf{kdom} \rightarrow_{\mathbf{PosU}} \mathbf{kcod}$  *below well founded* if there exists a function  $\underline{u} : \mathbf{kdom} \rightarrow \mathbf{Anti kcod}$  such that  $u = \uparrow \underline{u}$ .

We say that it is *finitely below well founded* (fBWF) if the supporting set is finite.

**Definition 12.4** (Above well founded – AWF)

We call a morphism  $\ell : \mathbf{kdom} \rightarrow_{\mathbf{PosL}} \mathbf{kcod}$  *above well founded* if there exists a function  $\bar{\ell} : \mathbf{kdom} \rightarrow \mathbf{Anti kcod}$  such that  $\ell = \downarrow \bar{\ell}$ . We say that it is *finitely above well founded* (fAWF) if the supporting set is finite.

Well-foundedness is a compositional property.

**Lemma 12.5** (Composition of well-founded maps in  $\mathbf{PosU}$  and  $\mathbf{PosL}$ ). Well-foundedness is compositional:

- If  $u_1$  and  $u_2$  are BWF (fBWF) then  $u_1 \circ u_2$  is BWF (fBWF).

- If  $\ell_1$  and  $\ell_2$  are AWF (fAWF) then  $\ell_1 \circ \ell_2$  is AWF (fAWF).

We call  $\mathbf{PosU}_w$  and  $\mathbf{PosL}_w$  the subcategories of well-founded morphisms in  $\mathbf{PosU}$  and  $\mathbf{PosL}$ . We call  $\mathbf{PosU}_f$  and  $\mathbf{PosL}_f$  the subcategories of finitely well-founded morphisms in  $\mathbf{PosU}$  and  $\mathbf{PosL}$ .

### 12.1.3. Well-foundedness of DPs

We extend the notion of well-foundedness to DPs.

**Definition 12.6** (Well-founded DP)

For a DP  $\mathbf{d} : \mathbf{F} \rightarrow_{\mathbf{DP}} \mathbf{R}$ , we say that it is:

- It is *forward (finitely) well-founded* if  $\mathbf{FR} \mathbf{d}$  is (finitely) well-founded.
- It is *backward (finitely) well-founded* if  $\mathbf{RF} \mathbf{d}$  is (finitely) well-founded.
- It is *bidirectionally (finitely) well-founded* if both  $\mathbf{FR} \mathbf{d}$  and  $\mathbf{RF} \mathbf{d}$  are (finitely) well-founded.

Also these properties are compositional.

**Lemma 12.7.** If  $\mathbf{d}_1$  and  $\mathbf{d}_2$  are (forward/backward/bidirectionally) (finitely) well-founded then  $\mathbf{d}_1 \circ \mathbf{d}_2$  is as well.

We call  $\mathbf{DP}_w$  ( $\mathbf{DP}_f$ ) the subcategories of bidirectionally (finitely) well-founded morphisms in  $\mathbf{DP}$ .

## 12.2. Lifting maps to DPs

A simple way to create a DP is to start with a monotone map  $g : \mathbf{P} \rightarrow_{\mathbf{Pos}} \mathbf{Q}$ .

There are two ways to do this, which we call the upper and lower lift  $\mathbf{DP\_LiftU}$  and  $\mathbf{DP\_LiftL}$ . In [1] these constructions are called *companion* and *conjoint*, respectively.

**Definition 12.8** (Upper lift  $\mathbf{DP\_LiftU}$ )

Given a monotone map  $g : \mathbf{P} \rightarrow_{\mathbf{Pos}} \mathbf{Q}$  we define the upper lift  $\mathbf{DP\_LiftU} g$  as:

$$\begin{aligned} \mathbf{DP\_LiftU} g : \mathbf{P} &\rightarrow_{\mathbf{DP}} \mathbf{Q} \\ \langle p^*, q \rangle &\longmapsto g(p^*) \leq_Q q \end{aligned} \tag{7}$$

**Definition 12.9** (Lower lift  $\mathbf{DP\_LiftL}$ )

Given a monotone map  $g : \mathbf{P} \rightarrow_{\mathbf{Pos}} \mathbf{Q}$  we define the lower lift  $\mathbf{DP\_LiftL} g$  as:

$$\begin{aligned} \mathbf{DP\_LiftL} g : \mathbf{Q} &\rightarrow_{\mathbf{DP}} \mathbf{P} \\ \langle q^*, p \rangle &\longmapsto q^* \leq_Q g(p) \end{aligned} \tag{8}$$

**Lemma 12.10.** For the lifted DP  $\mathbf{DP\_LiftL} g$  we have that

$$\begin{aligned} \mathbf{FR}(\mathbf{DP\_LiftL} g) : \mathbf{Q} &\rightarrow_{\mathbf{PosU}} \mathbf{P} \\ q^* &\longmapsto \{p \in \mathbf{P} \text{ such that } q^* \leq_Q g(p)\} \end{aligned} \tag{9}$$

and

$$\begin{aligned} \mathbf{RF}(\mathbf{DP\_LiftL} g) : \mathbf{P} &\rightarrow_{\mathbf{PosL}} \mathbf{Q} \\ p &\longmapsto \downarrow g(p) \end{aligned} \tag{10}$$

Note how  $\mathbf{RF}(\mathbf{DP\_LiftL} g)$  is easy to express because we can just apply  $g$  and take the lower closure. However the expression for  $\mathbf{FR}(\mathbf{DP\_LiftL} g)$  is more complicated and it has the flavor of an “inverse” operation on  $g$ .

When we look at the lifted DP  $\mathbf{DP\_LiftU} g$  we have the opposite situation.

**Lemma 12.11.** For the lifted DP  $\mathbf{DP\_LiftU} g : \mathbf{Q} \rightarrow_{\mathbf{DP}} \mathbf{P}$  we have that

$$\begin{aligned} \mathbf{FR}(\mathbf{DP\_LiftU} g) : \mathbf{Q} &\rightarrow_{\mathbf{PosU}} \mathbf{P} \\ q^* &\longmapsto \uparrow g(q^*) \end{aligned} \tag{11}$$

$$\begin{aligned} \text{RF}(\text{DP\_LiftU } g) : \mathbf{P} &\rightarrow_{\text{PosL}} \mathbf{Q} \\ p &\longmapsto \{q \in \mathbf{Q} \text{ such that } g(q) \leq_p p\} \end{aligned} \quad (12)$$

In this case we have that  $\text{FR}(\text{DP\_LiftU } g)$  is easy to express because we can just apply  $g$  and take the upper closure. However the expression for  $\text{RF}(\text{DP\_LiftU } g)$  is more complicated and it has the flavor of another “inverse” operation on  $g$ .

One of the computability questions we need to answer is under which condition on the map  $g$  we can represent the solutions of the  $\text{FR}$  and  $\text{RF}$  operators to produce upper/lower sets that can be represented by antichains.

We call these notions above/below well founded.

### 12.3. Upper and lower preimage of a monotone map

We can give a name to the inverse-like operations that appear in (9) and (12).

We call these operations the upper and lower pre-image of a monotone map.

**Definition 12.12** (Upper pre-image)

Given a monotone map  $g : \mathbf{Q} \rightarrow_{\text{Pos}} \mathbf{P}$  define the upper pre-image of  $g$  as:

$$\begin{aligned} \text{Ui } g : \mathbf{P} &\rightarrow_{\text{PosU}} \mathbf{Q} \\ p^* &\longmapsto \{q \in \mathbf{Q} \text{ such that } p^* \leq_p g(q)\} \end{aligned} \quad (13)$$

**Definition 12.13** (Lower pre-image)

Given a monotone map  $f : \mathbf{P} \rightarrow_{\text{Pos}} \mathbf{Q}$  define the lower pre-image of  $f$  as:

$$\begin{aligned} \text{Li } f : \mathbf{Q} &\rightarrow_{\text{PosL}} \mathbf{P} \\ q &\longmapsto \{p \in \mathbf{P} \text{ such that } f(p) \leq_q q\} \end{aligned} \quad (14)$$

**Example 12.14** (Example with  $\text{floor}$  and  $\text{ceil}$ ). We can compute the following:

$$\text{Ui } \text{ceil} : q \mapsto \{p \mid p > \text{floor}(q)\} = \uparrow \text{floor}(q) \quad (15)$$

$$\text{Li } \text{ceil} : q \mapsto \{p \mid p \leq \text{floor}(q)\} = \downarrow \text{floor}(q) \quad (16)$$

$$\text{Ui } \text{floor} : q \mapsto \{p \mid p \geq \text{ceil}(q)\} = \uparrow \text{ceil}(q) \quad (17)$$

$$\text{Li } \text{floor} : q \mapsto \{p \mid p < \text{ceil}(q)\} = \downarrow \text{ceil}(q) \quad (18)$$

Note that (17) and (16) given upper/lower sets that can be written as the upper/lower closure of a function, but (15) and (18) do not have this property.

**Lemma 12.15** (Contravariant functoriality).

$$\text{Li}(f \circ g) = \text{Li } g \circ \text{Li } f \quad (19)$$

$$\text{Ui}(f \circ g) = \text{Ui } g \circ \text{Ui } f \quad (20)$$

*Proof.* We prove the first equation. Let  $f : \mathbf{P} \rightarrow \mathbf{Q}$  and  $g : \mathbf{Q} \rightarrow \mathbf{R}$ . For any  $r \in \mathbf{R}$ :

$$[\text{Li}(f \circ g)](r) \doteq \{p \in \mathbf{P} \text{ such that } g(f(p)) \leq_r r\} \quad (21)$$

$$= \{p \in \mathbf{P} \text{ such that } f(p) \in [\text{Li } g](r)\} \quad (22)$$

$$= \bigcup_{q \in [\text{Li } g](r)} \{p \in \mathbf{P} \text{ such that } f(p) \leq_q q\} \quad (23)$$

$$= \bigcup_{q \in [\text{Li } g](r)} [\text{Li } f](q) \doteq [\text{Li } g \circ \text{Li } f](r) \quad (24)$$

The second equation follows similarly for the upper pre-image.  $\square$

## 12.4. Well-foundedness and Galois connections

If the map is invertible, then the upper and lower preimage are computed as the closure of the inverse of the function.

**Lemma 12.16.** If  $f : P \rightarrow_{\text{Pos}} Q$  is invertible, then

$$\mathbf{Li} f = \downarrow f^{-1} \quad (25)$$

$$\mathbf{Ui} f = \uparrow f^{-1} \quad (26)$$

*Proof.* For the lower pre-image:

$$[\mathbf{Li} f](q) \doteq \{p \in P \text{ such that } f(p) \leq_Q q\} \quad (27)$$

$$= \{p \in P \text{ such that } p \leq_P f^{-1}(q)\} \quad (\text{applying } f^{-1} \text{ to both sides}) \quad (28)$$

$$\doteq \downarrow f^{-1}(q) \quad (29)$$

For the upper pre-image:

$$[\mathbf{Ui} f](q) \doteq \{p \in Q \text{ such that } q \leq_Q f(p)\} \quad (30)$$

$$= \{p \in Q \text{ such that } f^{-1}(q) \leq_Q p\} \quad (\text{applying } f^{-1} \text{ to both sides}) \quad (31)$$

$$\doteq \uparrow f^{-1}(q) \quad (32)$$

□

Therefore, if the map is invertible, the upper and lower preimage are fBWF and fAWF.

We can relax the condition of invertibility and consider adjoint maps in the sense of Galois connections.

**Definition 12.17** (Galois connection)

A pair of maps  $f : P \rightarrow_{\text{Pos}} Q$ ,  $g : Q \rightarrow_{\text{Pos}} P$  is a Galois connection if  $\forall p \in P, \forall q \in Q$ ,

$$\frac{f(p) \leq_Q q}{p \leq_P g(q)}. \quad (33)$$

The map  $f$  is called “the lower<sup>a</sup> adjoint of  $g$ ” and  $g$  is called “the upper adjoint of  $f$ ”.

<sup>a</sup> $f$  is the “lower” because it appears in the left hand side of the condition (33).

**Example 12.18.** If  $f$  is invertible, the pair  $(f, f^{-1})$  is a Galois connection.

**Lemma 12.19.** If  $g : Q \rightarrow_{\text{Pos}} P$  has a lower adjoint  $f$ , then  $\mathbf{Ui} g = \uparrow f$ .

*Proof.* We compute  $\mathbf{Ui} g$  as:

$$\mathbf{Ui} g : p \mapsto \{q \in Q \text{ such that } p \leq_P g(q)\} \quad (34)$$

And using the property (33) we obtain

$$\{q \in Q \text{ such that } p \leq_P g(q)\} = \{q \in Q \text{ such that } f(p) \leq_Q q\} = \uparrow f(p) \quad (35)$$

□

**Lemma 12.20.** If  $f : P \rightarrow_{\text{Pos}} Q$  has an upper adjoint  $g$ , then  $\mathbf{Li} f = \downarrow g$ .

*Proof.* This is the dual of Lemma 12.19. □

Having an adjoint is a sufficient condition for the upper and lower preimage to be AWF and BWF, but it is not necessary. We give the following example.

**Lemma 12.21.** There are functions  $f$  such that  $\mathbf{Li} f$  is AWF, but  $f$  does not have an upper adjoint.

*Proof.* Consider the addition function on the natural numbers  $\text{add} : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ . We have

$$\mathbf{Li} \text{add} : n \mapsto \{(a, b) \text{ such that } a + b \leq n\} \quad (36)$$

For example:

$$\mathbf{Li\,add}(0) = \{\langle 0, 0 \rangle\} \quad (37)$$

$$\mathbf{Li\,add}(1) = \{\langle 0, 0 \rangle, \langle 0, 1 \rangle, \langle 1, 0 \rangle\} = \downarrow \{\langle 0, 1 \rangle, \langle 1, 0 \rangle\} \quad (38)$$

$$\mathbf{Li\,add}(2) = \downarrow \{\langle 0, 2 \rangle, \langle 1, 1 \rangle, \langle 2, 0 \rangle\} \quad (39)$$

Therefore,  $\mathbf{Li\,add}$  is AWF. However, there exists no upper adjoint of  $\mathbf{add}$ . A upper adjoint of  $\mathbf{add}$  would be a function  $\beta : \mathbb{N} \rightarrow \mathbb{N} \times \mathbb{N}$  such that

$$\forall \langle a, b \rangle \forall c \quad \mathbf{add}(\langle a, b \rangle) \leq c \Leftrightarrow \langle a, b \rangle \leq \beta(c) \quad (40)$$

This cannot work because the lower sets generated by  $\mathbf{Li\,add}$  are not principal. More formally, take  $c = 1$ . We have that

$$\forall \langle a, b \rangle \quad \mathbf{add}(\langle a, b \rangle) \leq 1 \Leftrightarrow \langle a, b \rangle \leq \beta(1) \quad (41)$$

The tuples that make the left hand side true are

$$\{\langle 0, 0 \rangle, \langle 0, 1 \rangle, \langle 1, 0 \rangle\} \quad (42)$$

but these cannot be written as  $\downarrow \beta(1)$  for any value that  $\beta$  can take.  $\square$

**Lemma 12.22.** There are functions  $g$  such that  $\mathbf{Ui\,g}$  is BWF, but  $g$  does not have a lower adjoint.

*Proof.* The function  $\mathbf{add}$  from the previous lemma works as an example.  $\square$

## 12.5. Well-foundedness and Scott-continuity

In the case that domain and codomain are dcpos, we can give a necessary and sufficient condition for the lower pre-image to be AWF: Scott continuity. In the other direction, the upper-pre-image is BWF if and only if the function is Scott-co-continuous.

### 12.5.1. Scott-continuity

**Definition 12.23** (Directed set)

A subset  $\mathbf{D} \subseteq \mathbf{A}$  is *directed* if it is non-empty and every pair  $a, b \in \mathbf{D}$  has an upper bound  $c \in \mathbf{D}$  ( $a \leq c$  and  $b \leq c$ ).

**Definition 12.24** (dcpo)

A directed complete partial order (dcpo) is a partial order in which every directed subset has a least upper bound.

We use the notation  $\bigvee^\uparrow \mathbf{D}$  to denote the supremum of a directed set  $\mathbf{D}$  in a dcpo.

**Definition 12.25** (Scott-continuity)

A monotone map  $f : \mathbf{P} \rightarrow_{\text{pos}} \mathbf{Q}$  between two dcpos is *Scott-continuous* if for every directed  $\mathbf{D} \subseteq \mathbf{P}$  the following equality holds:

$$f\left(\bigvee^\uparrow \mathbf{D}\right) = \bigvee^\uparrow f[\mathbf{D}].$$

(Note that because  $f$  is monotone,  $f[\mathbf{D}]$  is directed and because  $\mathbf{Q}$  is a dcpo, the supremum on the right always exists.)

It is useful to have a slightly broader version of Scott-continuity that is applicable to maps between arbitrary posets which might not be dcpos;

**Definition 12.26** (Pre-Scott-continuity)

A monotone map  $f : \mathbf{P} \rightarrow_{\text{pos}} \mathbf{Q}$  between two posets is *pre-Scott-continuous* if for every directed  $\mathbf{D} \subseteq \mathbf{P}$ , whenever  $\bigvee^\uparrow \mathbf{D}$  exists, then  $\bigvee^\uparrow f[\mathbf{D}]$  exists and

$$f\left(\bigvee^\uparrow \mathbf{D}\right) = \bigvee^\uparrow f[\mathbf{D}].$$

### 12.5.2. Co-Scott-continuity

We can define the dual notion of Scott-continuity that uses infima instead of suprema.



**Definition 12.27** (Filtered set)

A subset  $\mathbf{F} \subseteq \mathbf{Q}$  is *filtered* if it is non-empty and every pair  $a, b \in \mathbf{F}$  has a lower bound  $c \in \mathbf{F}$  ( $c \leq a$  and  $c \leq b$ ).

**Definition 12.28** (fcpo)

A filtered complete partial order (fcpo) is a partial order in which every filtered subset has a greatest lower bound.

We use the notation  $\bigwedge^\downarrow \mathbf{F}$  to denote the infimum of a filtered set  $\mathbf{F}$  in a fcpo.

**Definition 12.29** (Scott-co-continuity)

A monotone map  $g : \mathbf{Q} \rightarrow \mathbf{P}$  is *Scott-co-continuous* if for every filtered  $\mathbf{F} \subseteq \mathbf{Q}$  the following equality holds:

$$g\left(\bigwedge^\downarrow \mathbf{F}\right) = \bigwedge^\downarrow g[\mathbf{F}].$$

Because  $\mathbf{P}$  is a fcpo, the infimum on the right always exists.

**12.5.3. Scott-continuity and well-foundedness**

**Theorem 12.30.** If a monotone map  $f : \mathbf{P} \rightarrow \mathbf{Q}$  between two dcpos is Scott-continuous, then  $\mathbf{Lif} : \mathbf{Q} \rightarrow \mathbf{LP}$  is AWF. (Assumes the axiom of choice.)

*Proof.* Fixed a  $q$ , we already know that  $\mathbf{Lif}(q)$  is a lower set.

We can show that  $\mathbf{Lif}(q)$  is closed under directed suprema. Let  $\mathbf{D} \subseteq \mathbf{Lif}(q)$  be directed. Because  $\mathbf{P}$  is a dcpo,  $s = \sup \mathbf{D}$  exists. Scott continuity gives  $f(s) = \sup f[\mathbf{D}] \leq q$ , so  $s \in \mathbf{Lif}(q)$ .

So, every chain in  $\mathbf{Lif}(q)$  is directed and has an upper bound in  $\mathbf{Lif}(q)$ .

By Zorn's lemma (Lemma 12.32),  $\mathbf{Lif}(q)$  possesses maximal elements; let  $\text{Max}(\mathbf{Lif}(q))$  denote their set, obviously an antichain.

Because  $\mathbf{Lif}(q)$  is a lower set, every  $p \in \mathbf{Lif}(q)$  lies below some maximal element in  $\text{Max}(\mathbf{Lif}(q))$ , by applying Zorn's lemma again to  $L_{\geq p} \doteq \{x \in \mathbf{Lif}(q) \mid p \leq x\}$ . Hence

$$\mathbf{Lif}(q) = \downarrow \text{Max}(\mathbf{Lif}(q)), \quad (43)$$

which witnesses the AWF property.  $\square$

**Theorem 12.31.** For a monotone map  $f : \mathbf{P} \rightarrow \mathbf{Q}$  between two dcpos, if  $\mathbf{Lif} : \mathbf{Q} \rightarrow \mathbf{LP}$  is finitely above well founded (fAWF) then  $f$  is Scott-continuous. (Assumes the axiom of choice.)

*Proof.* Choose any directed  $\mathbf{D} \subseteq \mathbf{P}$  and call

$$\bar{d} \doteq \bigvee^\uparrow \mathbf{D} \quad (44)$$

its supremum, which exists because  $\mathbf{D}$  is directed and  $\mathbf{P}$  is a dcpo. Consider  $f[\mathbf{D}]$ , the image of  $\mathbf{D}$  under  $f$ . Because  $f$  is monotone,  $f[\mathbf{D}]$  is directed. Because  $\mathbf{Q}$  is a dcpo, the supremum of  $f[\mathbf{D}]$  exists. Call it

$$\bar{q} \doteq \bigvee^\uparrow f[\mathbf{D}]. \quad (45)$$

To prove that  $f$  is Scott-continuous, we need to show that  $f(\bar{d}) = \bar{q}$ .

Because of monotonicity we have that

$$f(\bar{d}) \geq_Q \bar{q}. \quad (46)$$

So our goal is to prove  $f(\bar{d}) \leq_Q \bar{q}$ ; equality will follow from (46).

Proof by contradiction: assume that

$$f(\bar{d}) \not\leq_Q \bar{q} \quad (\text{by contradiction}) \quad (47)$$

Define

$$\mathbf{L} := (\mathbf{Lif})(\bar{q}) \quad (48)$$

to be the lower pre-image of  $\bar{q}$ . Because of (47) we have that

$$\bar{d} \notin \mathbf{L}. \quad (49)$$

At the same time, we have  $\mathbf{D} \subseteq \mathbf{L}$ , because for any  $x \in \mathbf{D}$ :

$$f(x) \leq_Q \bigvee^{\uparrow} f[\mathbf{D}] = \bar{q}. \quad (50)$$

Because  $\mathbf{L} \mathbf{i} f$  is fAWF there exists a *finite* antichain  $\mathbf{S}$  such that  $\mathbf{L} = \downarrow \mathbf{S}$ .

To summarize so far, the assumption (47) allowed us to have that

$$\mathbf{D} \subseteq \mathbf{L} = \downarrow \mathbf{S} \quad (51)$$

yet, because of (49) we have that

$$\bar{d} \notin \downarrow \mathbf{S}. \quad (52)$$

Because  $\mathbf{S}$  is finite,  $\mathbf{D}$  is directed, and  $\mathbf{D} \subseteq \downarrow \mathbf{S}$ , by Lemma 12.33 we have that  $\mathbf{D}$  has an upper bound in  $s_D \in \mathbf{S}$ . Because  $\bar{d}$  is the least upper bound of  $\mathbf{D}$  we have that necessarily  $\bar{d} \leq_A s_D$ . Then we have that

$$\bar{d} \in \downarrow \mathbf{S}, \quad (53)$$

which contradicts (52). Therefore, our assumption (47) is impossible, and we conclude that  $f(\bar{d}) \leq_Q \bar{q}$ . Together with (46) we have that  $f(\bar{d}) = \bar{q}$ .

Because the set  $\mathbf{D}$  was arbitrary, we have shown that for any directed  $\mathbf{D} \subseteq \mathbf{A}$  we have that

$$f\left(\bigvee^{\uparrow} \mathbf{D}\right) = \bigvee^{\uparrow} f[\mathbf{D}]. \quad (54)$$

Therefore,  $f$  is Scott-continuous. □

**Lemma 12.32** (Zorn's lemma). Every non-empty poset in which every chain has an upper bound has a maximal element.

*This is equivalent to the axiom of choice.*

**Lemma 12.33** (Single-pigeon-dropping principle for antichains). Let  $\mathbf{S}$  be a *finite* antichain in a poset  $\mathbf{P}$ . If a (finite or infinite) directed set  $\mathbf{D}$  is such that  $\mathbf{D} \subseteq \downarrow \mathbf{S}$ , then  $\mathbf{D}$  has an upper bound in  $\mathbf{S}$ . (The proof uses the axiom of choice.)

*Proof.* By contradiction. Assume that  $\mathbf{D}$  has no upper bound in  $\mathbf{S}$ . Then for each  $s \in \mathbf{S}$  we can choose (using the axiom of choice) an element  $c_s \in \mathbf{D}$  such that

$$c_s \not\leq s. \quad (55)$$

Call  $\mathbf{C} = \{c_s \mid s \in \mathbf{S}\}$  the set of all these counter-examples. Since  $\mathbf{S}$  is finite,  $\mathbf{C}$  is also finite. Because  $\mathbf{D}$  is directed,  $\mathbf{C} \subseteq \mathbf{D}$ , and  $\mathbf{C}$  is finite,  $\mathbf{C}$  has an upper bound  $\bar{c} \in \mathbf{D}$ , which can be constructed by finitely iterating the directedness property. (This part would not work if  $\mathbf{C}$  was infinite.) Because  $\mathbf{D} \subseteq \downarrow \mathbf{S}$ , we have that  $\bar{c} \in \downarrow \mathbf{S}$ , which means that there exists  $s_C \in \mathbf{S}$  such that  $\bar{c} \leq s_C$ . Therefore, for all  $s$ ,  $c_s \leq \bar{c} \leq s_C$ . In particular for  $s = s_C$ , we have that  $c_{s_C} \leq s_C$ . But this contradicts (55). □

Finally, we state the dual of the previous results.

**Theorem 12.34.** If a monotone map  $f : \mathbf{Q} \rightarrow \mathbf{P}$  between two fcpos is Scott co-continuous, then  $\mathbf{U} \mathbf{i} f$  is BWF.

**Theorem 12.35.** If  $\mathbf{U} \mathbf{i} g$  is fBWF, then  $g$  is Scott co-continuous.

## 12.6. Lifting as functors

We can see that  $\mathbf{DP\_LiftL}$  and  $\mathbf{DP\_LiftU}$  are monotone functors.

**Lemma 12.36** (Monotonicity of lifting). The contravariant functor  $\mathbf{DP\_LiftL} : \mathbf{Pos}^{\text{op}} \rightarrow \mathbf{DP}$  is monotone. The covariant functor  $\mathbf{DP\_LiftU} : \mathbf{Pos} \rightarrow \mathbf{DP}$  is antitone.

*Proof.* Proof of antitonicity of  $\mathbf{DP\_LiftU}$ : let  $f, g : \mathbf{P} \rightarrow_{\mathbf{Pos}} \mathbf{Q}$  and  $f \leq_{\mathbf{Pos}} g$ . We check when they are feasible:

$$\mathbf{DP\_LiftU}(f)(p, q) = f(p) \leq_Q q \quad (56)$$

$$\mathbf{DP\_LiftU}(g)(p, q) = g(p) \leq_Q q \quad (57)$$

Because  $f(p) \leq_Q g(p)$ , we have that  $\mathbf{DP\_LiftU}(g)(p, q) \Rightarrow \mathbf{DP\_LiftU}(f)(p, q)$ , which means that  $\mathbf{DP\_LiftU}(f)$  is more feasible than

$\text{DP\_LiftU}(g)$ . Thus we have that  $\text{DP\_LiftU}$  is antitone:

$$\frac{f \leq_{\mathbf{Pos}} g}{\text{DP\_LiftU}(g) \leq_{\mathbf{DP}} \text{DP\_LiftU}(f)} . \quad (58)$$

The proof of monotonicity of  $\text{DP\_LiftL}$  is similar. □

### 12.6.1. Restriction to Scott-(co)continuous maps

We can now look at the restriction of  $\text{DP\_LiftL}$  and  $\text{DP\_LiftU}$  to Scott-(co)continuous maps.

**Definition 12.37 (DCPO)**

**DCPO** is the subcategory of **Pos** that has  $\text{dcpos}$  as objects and Scott-continuous maps as morphisms.

**Definition 12.38 (FCPO)**

**FCPO** is the subcategory of **Pos** that has  $\text{fcpos}$  as objects and Scott-co-continuous maps as morphisms.

**Lemma 12.39.** The restrictions of  $\text{DP\_LiftU}$  and  $\text{DP\_LiftL}$  to (co)Scott-continuous maps are functors into the subcategory of well-founded DPs:

$$\text{DP\_LiftU} : \mathbf{DCPO} \rightarrow \mathbf{DP}_w \quad (59)$$

and a functor

$$\text{DP\_LiftL} : \mathbf{FCPO}^{\text{op}} \rightarrow \mathbf{DP}_w \quad (60)$$

## 13. Scalable computation for DPs

### 13.1. Scalable maps

We define two categories  $\mathbf{SPosL}$  and  $\mathbf{SPosU}$  that are generalizations of the categories  $\mathbf{PosL}$  and  $\mathbf{PosU}$ .

Each morphism in  $\mathbf{SPosL}$  has associated a pair of “resolution” posets, and two maps into  $\mathbf{PosL}$ , which are meant to be the “optimistic” and “pessimistic” version of the map, thus representing an interval in  $\mathbf{PosL}$ .

**Definition 13.1**

Given three posets  $k\text{dom}$ ,  $k\text{cod}$ , and a pair of posets  $S = \langle S^{\ominus}, S^{\oplus} \rangle$ , referred to as the “optimistic” and “pessimistic” “resolution” posets, a morphism in  $\mathbf{SPosL}$

$$sl : \{S\} k\text{dom} \rightarrow_{\mathbf{SPosL}} k\text{cod} \quad (1)$$

is defined by giving two maps

$$sl^{\oplus} : S^{\oplus} \rightarrow_{\mathbf{Pos}} (k\text{dom} \rightarrow_{\mathbf{PosL}} k\text{cod}) \quad (2)$$

$$sl^{\ominus} : S^{\ominus} \rightarrow_{\mathbf{Pos}} (k\text{dom} \rightarrow_{\mathbf{PosL}} k\text{cod}) \quad (3)$$

that satisfy the following condition:

$$\forall o \in S^{\oplus} : \forall p \in S^{\ominus} : sl^{\oplus}(p) \leq_{\mathbf{PosL}} sl^{\ominus}(o) \quad (4)$$

**Definition 13.2**

Given three posets  $k\text{dom}$ ,  $k\text{cod}$ , and a pair of posets  $S = \langle S^{\ominus}, S^{\oplus} \rangle$ , referred to as the “optimistic” and “pessimistic” “resolution” posets, a morphism in  $\mathbf{SPosU}$

$$su : \{S\} k\text{dom} \rightarrow_{\mathbf{SPosU}} k\text{cod} \quad (5)$$

is defined by giving two maps

$$su^{\oplus} : S^{\oplus} \rightarrow_{\mathbf{Pos}} (k\text{dom} \rightarrow_{\mathbf{PosU}} k\text{cod}) \quad (6)$$

$$su^{\ominus} : S^{\ominus} \rightarrow_{\mathbf{Pos}} (k\text{dom} \rightarrow_{\mathbf{PosU}} k\text{cod}) \quad (7)$$

that satisfy the following condition:

$$\forall o \in S^{\oplus} : \forall p \in S^{\ominus} : su^{\oplus}(p) \leq_{\mathbf{PosU}} su^{\ominus}(o) \quad (8)$$

**Lemma 13.3.**  $\mathbf{SPosU}$  and  $\mathbf{SPosL}$  are traced monoidal categories.

The identities and series compositions for these categories are reported in Chapter 22.

**Remark 13.4.** If  $S^{\oplus}$  and  $S^{\ominus}$  were the same poset  $S$ , then a morphism of  $\mathbf{SPosU}$  would be equivalent to a monotone function from  $R$  to the poset of intervals

$$su : S \rightarrow_{\mathbf{Pos}} \text{P\_C\_Twisted}(k\text{dom} \rightarrow_{\mathbf{PosU}} k\text{cod}). \quad (9)$$

We choose to consider explicitly the case where  $S^{\oplus}$  and  $S^{\ominus}$  are different posets, as it is useful in certain cases, such as when we have a different number of optimistic and pessimistic approximations.

### 13.2. Approximation of DP queries

**Definition 13.5** (Approximation of DP forward queries)

Given a DP  $d : \mathbf{F} \rightarrow_{\mathbf{DP}} \mathbf{R}$ , we define the following admissible sets of query solutions:

- $\text{FR}_f^{\checkmark} d$  is the subset of  $\mathbf{SPosU}_f(\mathbf{F}, \mathbf{R})$  containing all morphisms  $su$  such that

$$\forall p \in S^{\oplus} : \forall o \in S^{\ominus} : su^{\oplus}(p) \leq (\text{FR } d) \leq su^{\ominus}(o) \quad (10)$$

These are the scalable computable solutions that are consistent with the actual solution. These are not guaranteed to get close to the actual solution.

- $FR_f^\ominus \mathbf{d}$  is the subset of  $\mathbf{SPosU}_f(\mathbf{F}, \mathbf{R})$  containing all morphisms  $su$  such that

$$\inf_{o \in S^\ominus} su^\ominus(o) \leq (FR \mathbf{d}) \quad (11)$$

These are the scalable computable solutions that tend to a pessimistic solution.

- $FR_f^\oplus \mathbf{d}$  is the subset of  $\mathbf{SPosU}_f(\mathbf{F}, \mathbf{R})$  containing all morphisms  $su$  such that

$$(FR \mathbf{d}) \leq \sup_{p \in S^\oplus} su^\oplus(p) \quad (12)$$

These are the scalable computable solutions that tend to an optimistic solution.

- $FR_f^\star \mathbf{d}$  is the subset of  $\mathbf{SPosU}_f(\mathbf{F}, \mathbf{R})$  containing all morphisms  $su$  such that

$$\sup_{p \in S^\oplus} su^\oplus(p) \simeq (FR \mathbf{d}) \simeq \inf_{o \in S^\ominus} su^\ominus(o) \quad (13)$$

These are the scalable computable solutions that tend to actual solution. It is the intersection of the two previous sets:

$$FR_f^\star \mathbf{d} = (FR_f^\ominus \mathbf{d}) \cap (FR_f^\oplus \mathbf{d}) \quad (14)$$

Note that the infimum/supremum are not necessarily attained as part of the set.

**Lemma 13.6** (Compositionality of forward queries approximations). The following holds:

$$\begin{array}{c} \frac{su_1 \in (FR_f^\vee \mathbf{d}_1) \quad su_2 \in (FR_f^\vee \mathbf{d}_2)}{su_1 \circ su_2 \in FR_f^\vee(\mathbf{d}_1 \circ \mathbf{d}_2)} \quad \frac{su_1 \in (FR_f^\star \mathbf{d}_1) \quad su_2 \in (FR_f^\star \mathbf{d}_2)}{su_1 \circ su_2 \in FR_f^\star(\mathbf{d}_1 \circ \mathbf{d}_2)} \\ \\ \frac{su_1 \in (FR_f^\ominus \mathbf{d}_1) \quad su_2 \in (FR_f^\ominus \mathbf{d}_2)}{su_1 \circ su_2 \in FR_f^\ominus(\mathbf{d}_1 \circ \mathbf{d}_2)} \quad \frac{su_1 \in (FR_f^\oplus \mathbf{d}_1) \quad su_2 \in (FR_f^\oplus \mathbf{d}_2)}{su_1 \circ su_2 \in FR_f^\oplus(\mathbf{d}_1 \circ \mathbf{d}_2)} \end{array} \quad (15)$$

Another way to write the previous lemma is:

$$(FR_f^\vee \mathbf{d}_1) \circ (FR_f^\vee \mathbf{d}_2) \subseteq FR_f^\vee(\mathbf{d}_1 \circ \mathbf{d}_2) \quad (16)$$

$$(FR_f^\star \mathbf{d}_1) \circ (FR_f^\star \mathbf{d}_2) \subseteq FR_f^\star(\mathbf{d}_1 \circ \mathbf{d}_2) \quad (17)$$

$$(FR_f^\ominus \mathbf{d}_1) \circ (FR_f^\ominus \mathbf{d}_2) \subseteq FR_f^\ominus(\mathbf{d}_1 \circ \mathbf{d}_2) \quad (18)$$

$$(FR_f^\oplus \mathbf{d}_1) \circ (FR_f^\oplus \mathbf{d}_2) \subseteq FR_f^\oplus(\mathbf{d}_1 \circ \mathbf{d}_2) \quad (19)$$

We repeat the construction for backward queries.

**Definition 13.7** (Approximation of DP backward queries)

Given a DP  $\mathbf{d} : \mathbf{F} \rightarrow_{\text{DP}} \mathbf{R}$ , we define the following admissible sets of query solutions:

- $RF_f^\vee \mathbf{d}$  is the subset of  $\mathbf{SPosL}_f(\mathbf{R}, \mathbf{F})$  containing all morphisms  $sl$  such that

$$\forall p \in S^\ominus : \quad \forall o \in S^\oplus : \quad sl^\ominus(p) \leq (RF \mathbf{d}) \leq sl^\oplus(o) \quad (20)$$

- $RF_f^\ominus \mathbf{d}$  is the subset of  $\mathbf{SPosL}_f(\mathbf{R}, \mathbf{F})$  containing all morphisms  $sl$  such that

$$\inf_{o \in S^\oplus} sl^\oplus(o) \leq (RF \mathbf{d}) \quad (21)$$

- $RF_f^\oplus \mathbf{d}$  is the subset of  $\mathbf{SPosL}_f(\mathbf{R}, \mathbf{F})$  containing all morphisms  $sl$  such that

$$(RF \mathbf{d}) \leq \sup_{p \in S^\ominus} sl^\ominus(p) \quad (22)$$

- $RF_f^\star \mathbf{d}$  is the subset of  $\mathbf{SPosL}_f(\mathbf{R}, \mathbf{F})$  containing all morphisms  $sl$  such that

$$\sup_{p \in S^\ominus} sl^\ominus(p) \simeq (RF \mathbf{d}) \simeq \inf_{o \in S^\oplus} sl^\oplus(o) \quad (23)$$

These are the scalable computable solutions that tend to actual solution. It is the intersection of the two previous sets:

$$RF_f^\star \mathbf{d} = (RF_f^\ominus \mathbf{d}) \cap (RF_f^\oplus \mathbf{d}) \quad (24)$$

Note that the infimum/supremum are not necessarily attained as part of the set.

**Lemma 13.8** (Compositionality of backward queries approximations). The following holds:

$$\begin{array}{c} \frac{sl_1 \in (RF_f^\vee \mathbf{d}_1) \quad sl_2 \in (RF_f^\vee \mathbf{d}_2)}{sl_2 \circ sl_1 \in RF_f^\vee(\mathbf{d}_1 \circ \mathbf{d}_2)} \quad \frac{sl_1 \in (RF_f^\star \mathbf{d}_1) \quad sl_2 \in (RF_f^\star \mathbf{d}_2)}{sl_2 \circ sl_1 \in RF_f^\star(\mathbf{d}_1 \circ \mathbf{d}_2)} \\[10pt] \frac{sl_1 \in (RF_f^\ominus \mathbf{d}_1) \quad sl_2 \in (RF_f^\ominus \mathbf{d}_2)}{sl_2 \circ sl_1 \in RF_f^\ominus(\mathbf{d}_1 \circ \mathbf{d}_2)} \quad \frac{sl_1 \in (RF_f^\oplus \mathbf{d}_1) \quad sl_2 \in (RF_f^\oplus \mathbf{d}_2)}{sl_2 \circ sl_1 \in RF_f^\oplus(\mathbf{d}_1 \circ \mathbf{d}_2)} \end{array} \quad (25)$$

Note that because of contravariance, the order of the morphisms is reversed for the backward queries ( $sl_2 \circ sl_1$  instead of  $su_1 \circ su_2$ ).

Similar results hold for the other types of morphisms composition in **DP**: monoidal product, trace, union, and intersection.

## 14. Design problems with implementations (DPIs)

### 14.1. DPIs

#### Definition 14.1

A design problem with implementation and blueprints (“DPIB” or simply “DPI”)

$$\mathbf{d} : \mathbf{F} \rightarrow_{\mathbf{DPI}} \mathbf{R}\{\mathcal{B}\} \quad (1)$$

is defined by the following data:

- A poset  $\mathbf{F}$  of “functionalities”
- A poset  $\mathbf{R}$  of “requirements”
- A poset  $\mathbf{I}$  of “implementations”
- A poset  $\mathcal{B}$  of “blueprints”
- A monotone map  $\text{prov} : \mathbf{I} \rightarrow_{\text{Pos}} \mathbf{F}$
- A monotone map  $\text{req} : \mathbf{I} \rightarrow_{\text{Pos}} \mathbf{R}^{\text{op}}$
- A monotone map  $\text{avail} : \mathbf{I}^{\text{op}} \rightarrow_{\text{Pos}} \mathbf{Bool}$
- A monotone map  $\text{feas} : \mathbf{I} \rightarrow_{\text{Pos}} \mathbf{Bool}$
- A monotone map  $\text{IB} : \mathbf{I}/(\text{avail} \wedge \text{feas}) \rightarrow_{\text{Pos}} \mathcal{B}$

where  $\mathbf{I}/(\text{avail} \wedge \text{feas})$  is the subset of  $\mathbf{I}$  such that  $\text{avail}$  and  $\text{feas}$  are true.

*This construction is described by the schema DP (Section 26.12).*

**Interpretation of the implementation space** The implementation space  $\mathbf{I}$  are the decision variables. An implementation  $i$  defines univocally the functionality and requirements by the map  $\text{prov}$  and  $\text{req}$ .

The two maps  $\text{avail}$  and  $\text{feas}$  together denote the feasible subset of implementations. Because  $\text{avail}$ ’s domain is  $\mathbf{I}^{\text{op}}$ , it represents a lower set; because  $\text{feas}$ ’s domain is  $\mathbf{I}$ , it represents an upper set. This ensures that, by construction, all DPIs have feasible implementations in the intersection of an upper and a lower set. Moreover, because we construct these two functions explicitly for all constructions, we can say that the problem of deciding whether an implementation is feasible is a *decidable* problem.

The order on  $\mathbf{I}$  represents a preference structure on the implementations in addition to their external properties of functionality/requirements given by  $\text{prov}$  and  $\text{req}$ . We adopt the convention that “smaller is better”. So  $\text{avail}$  tells us that higher implementations might not be available, while  $\text{feas}$  tells us that lower implementations might not be feasible. We will see that  $\text{feas}$  constraints derive from the composition of DPIs.

**Interpretation of the blueprint space** The poset  $\mathcal{B}$  represents the “actionable information” from the implementation. For example, in the design of a robot, the blueprint space could be the bill of materials, while the implementation space contains more details including the values of voltage and current on each wire. A blueprint is entirely determined by the implementation by the map  $\text{IB}$ .

We also think of the blueprint space as the “public result” from the design problem, while the implementation space is the “private data”. We will consider a notion of congruence between DPIs if they give the same blueprint results, even if the internal implementation details are different. This simple concept will allow us to reason about DPI “simplifications” and “optimizations”; the compiler tries to find the DPI whose queries are simplest to compute among all the possible implementations.

For this reason, when we write the type of a DPI as

$$\mathbf{dp} : \mathbf{F} \rightarrow_{\mathbf{DPI}} \mathbf{R}\{\mathcal{B}\}, \quad (2)$$

we specify the blueprint space  $\mathcal{B}$ , which, along with the functionality space  $\mathbf{F}$  and the requirement space  $\mathbf{R}$ , is the “public interface” of the DPI, but we do not specify the implementation space  $\mathbf{I}$ .

**Remark 14.2.** The DPI construction here is a generalization of the definition of DPI [1] in which:

- The implementation space  $\mathbf{I}$  here is a poset, not a set.
- We added the blueprint space  $\mathcal{B}$ .

- We made the availability and feasibility functions explicit.

These are the benefits:

- By construction the subset of valid implementations (those who are “available” and “feasible”) is decidable, as we construct the indicator functions directly.
- By allowing a distinction between the implementation and blueprint spaces, we can distinguish about the “external interfaces”, which contains only  $\mathcal{B}$  and the “internal” implementation. In such a way we can study equivalence between DPIs and approaches for “simplification” that change the internal implementation without affecting the external interface.

## 14.2. Optimization queries associated to a DPI

A DPI represent a *model* of a design problem. The “queries” are the “questions” that the user can ask about the design problem.

As for a DP, we are primarily interested in two quantitative questions for a DPI:

1. Given a minimum functionality, what are the minimal resources required?
2. Given a maximum budget, what is the best functionality that can be achieved?

These questions are duals to each other.

Because DPIs have additional structure in implementations and blueprints, we have for each query three variants:

1. Only ask about functionality/requirements.
2. Also derive the implementations
3. Also derive the blueprints

The following are the definitions of the forward and backward queries when we are interested in recovering the implementations.

### Definition 14.3 (FixFunMinReqI)

Given a DPI  $\mathbf{dp}$  and a functionality  $f_0^*$ , we call  $\text{FixFunMinReqI}^{\mathbf{dp}}(f_0^*)$  the optimization problem

$$\text{using } i \in \mathbf{I}, \quad (3)$$

$$r \in \mathbf{R}, \quad (4)$$

$$\text{Min}_{\mathbf{R}} r, \text{ then lexicographically Min}_{\mathbf{I}} i, \quad (5)$$

$$\text{such that } \text{avail}(i) = \top, \quad (6)$$

$$\text{feas}(i) = \top, \quad (7)$$

$$f_0^* \leq_{\mathbf{F}} \text{prov}(i), \quad (8)$$

$$\text{req}(i) \leq_{\mathbf{R}} r. \quad (9)$$

Note that the objective function is multidimensional: because  $\mathbf{R}$  is a poset, the objective function uses Min rather than min: there could be a pareto frontier of solutions. The objective is also lexicographically ordered: first we want to minimize the requirements, then we want to choose the “simplest” implementations.

The following is the dual problem, where we fix the requirements and we want to maximize the functionality.

### Definition 14.4 (FixReqMaxFunI)

Given a DPI  $\mathbf{dp}$  and a requirement value  $r_0$ , we call  $\text{FixReqMaxFunI}^{\mathbf{dp}}(r_0)$  the optimization problem

$$\text{using } i \in \mathbf{I}, \quad (10)$$

$$f \in \mathbf{F}, \quad (11)$$

$$\text{Max}_{\mathbf{F}} f, \text{ then lexicographically Min}_{\mathbf{I}} i, \quad (12)$$

$$\text{such that } \text{avail}(i) = \top, \quad (13)$$

$$\text{feas}(i) = \top, \quad (14)$$

$$f \leq_{\mathbf{F}} \text{prov}(i), \quad (15)$$

$$\text{req}(i) \leq_{\mathbf{R}} r_0. \quad (16)$$

The following are the variations of the queries when we are interested in recovering the blueprints.



**Definition 14.5** (FixFunMinReqB)

Given a DPI  $\mathbf{dp}$  and a functionality  $f_0^*$ , we call  $\text{FixFunMinReqB}^{\mathbf{dp}}(f_0^*)$  the optimization problem

$$\text{using } i \in \mathbf{I}, \quad (17)$$

$$r \in \mathbf{R}, \quad (18)$$

$$b \in \mathbf{B}, \quad (19)$$

$$\text{Min}_{\mathbf{R}} r, \text{ then lexicographically Min}_{\mathbf{B}} b, \quad (20)$$

$$\text{such that } \text{avail}(i) = \top, \quad (21)$$

$$\text{feas}(i) = \top, \quad (22)$$

$$f_0^* \leq_{\mathbf{F}} \text{prov}(i), \quad (23)$$

$$\text{req}(i) \leq_{\mathbf{R}} r, \quad (24)$$

$$\text{IB}(i) \leq_{\mathbf{B}} b. \quad (25)$$

**Definition 14.6** (FixReqMaxFunB)

Given a DPI  $\mathbf{dp}$  and a requirement value  $r_0$ , we call  $\text{FixReqMaxFunB}^{\mathbf{dp}}(r_0)$  the optimization problem

$$\text{using } i \in \mathbf{I}, \quad (26)$$

$$f \in \mathbf{F}, \quad (27)$$

$$b \in \mathbf{B}, \quad (28)$$

$$\text{Max}_{\mathbf{F}} f, \text{ then lexicographically Min}_{\mathbf{B}} b, \quad (29)$$

$$\text{such that } \text{avail}(i) = \top, \quad (30)$$

$$\text{feas}(i) = \top, \quad (31)$$

$$f \leq_{\mathbf{F}} \text{prov}(i), \quad (32)$$

$$\text{req}(i) \leq_{\mathbf{R}} r_0 \quad (33)$$

$$\text{IB}(i) \leq_{\mathbf{B}} b. \quad (34)$$

As in the case of DP, these problems are not necessarily well-posed: there could be feasible solutions but the feasible set could be not well-founded.

### 14.3. Categories $\text{PosUI}$ and $\text{PosLI}$

We now define two categories,  $\text{PosLI}$  and  $\text{PosUI}$ , which are generalizations of  $\text{PosL}$  and  $\text{PosU}$  whose morphisms carry the implementation information.

**Definition 14.7** (Morphisms of  $\text{PosUI}$ )

Given three posets  $\mathbf{kdom}$ ,  $\mathbf{kcod}$ , and  $\mathbf{kimp}$  a morphism

$$u : \mathbf{kdom} \rightarrow_{\text{PosUI}} \mathbf{kcod} \{\mathbf{kimp}\} \quad (35)$$

is a monotone map

$$u : \mathbf{kdom}^{\text{op}} \rightarrow_{\text{Pos}} \text{P\_C\_UpperSets}(\text{P\_C\_Lexicographic}(\llbracket \mathbf{kcod}, \mathbf{kimp} \rrbracket)) \quad (36)$$

*This construction is described by the schema UMap (Section 26.7).*

Note the  $\mathbf{kdom}^{\text{op}}$  in the domain: as the required functionality increases, the feasible requirements decrease.

**Definition 14.8** (Morphisms of  $\text{PosLI}$ )

Given three posets  $\mathbf{kdom}$ ,  $\mathbf{kcod}$ , and  $\mathbf{kimp}$  a morphism

$$\ell : \mathbf{kdom} \rightarrow_{\text{PosLI}} \mathbf{kcod} \{\mathbf{kimp}\} \quad (37)$$

is a monotone map

$$\ell : \mathbf{kdom} \rightarrow_{\text{Pos}} \text{P\_C\_LowerSets}(\text{P\_C\_Lexicographic}(\llbracket \mathbf{kcod}, \mathbf{kimp}^{\text{op}} \rrbracket)) \quad (38)$$

*This construction is described by the schema LMap (Section 26.6).*

Note that there is a  $\text{kimp}^{\text{op}}$  in the codomain: we want to maximize functionality in  $\text{kcod}$  but still minimize the implementation in  $\text{kimp}$ . The identity and series constructions are reported in Chapter 21.

### 14.3.1. Relating $\text{PosUI}$ and $\text{PosLI}$ to $\text{PosU}$ and $\text{PosL}$

**Definition 14.9** (Projection)

Given a morphism  $u : \text{kdom} \rightarrow_{\text{PosUI}} \text{kcod} \{\text{kimp}\}$ , we define the projection

$$\begin{aligned} \text{proj}(u) : \text{kdom} &\rightarrow_{\text{PosU}} \text{kcod} \\ f^* &\longmapsto \{r \text{ for } \langle r, i \rangle \in u(f^*)\} \end{aligned} \quad (39)$$

**Definition 14.10** (Projection)

Given a morphism  $\ell : \text{kdom} \rightarrow_{\text{PosLI}} \text{kcod} \{\text{kimp}\}$ , we define the projection

$$\begin{aligned} \text{proj}(\ell) : \text{kdom} &\rightarrow_{\text{PosL}} \text{kcod} \\ r &\longmapsto \{f \text{ for } \langle f, i \rangle \in \ell(r)\} \end{aligned} \quad (40)$$

**Lemma 14.11.**

$$\text{proj}(u_1 \circ u_2) = \text{proj}(u_1) \circ \text{proj}(u_2) \quad (41)$$

$$\text{proj}(\ell_1 \circ \ell_2) = \text{proj}(\ell_1) \circ \text{proj}(\ell_2) \quad (42)$$

**Definition 14.12** (Embedding)

Given a morphism  $u : \text{kdom} \rightarrow_{\text{PosU}} \text{kcod}$ , we define the embedding

$$\begin{aligned} \text{embed}(u) : \text{kdom} &\rightarrow_{\text{PosUI}} \text{kcod} \{\mathbf{I}\} \\ f^* &\longmapsto \{\langle r, * \rangle \text{ for } r \in u(f^*)\} \end{aligned} \quad (43)$$

**Definition 14.13** (Embedding)

Given a morphism  $\ell : \text{kdom} \rightarrow_{\text{PosL}} \text{kcod}$ , we define the embedding

$$\begin{aligned} \text{embed}(\ell) : \text{kdom} &\rightarrow_{\text{PosLI}} \text{kcod} \{\mathbf{I}\} \\ r &\longmapsto \{\langle f, * \rangle \text{ for } f \in \ell(r)\} \end{aligned} \quad (44)$$

**Lemma 14.14.**

$$\text{embed}(u_1 \circ u_2) = \text{embed}(u_1) \circ \text{embed}(u_2) \quad (45)$$

$$\text{embed}(\ell_1 \circ \ell_2) = \text{embed}(\ell_1) \circ \text{embed}(\ell_2) \quad (46)$$

**Lemma 14.15.**

$$\text{proj}(\text{embed } u) = u \quad (47)$$

$$\text{proj}(\text{embed } \ell) = \ell \quad (48)$$

### 14.3.2. Pre-order on $\text{PosLI}$ and $\text{PosUI}$

**Definition 14.16** (Pre-order on  $\text{PosLI}$  and  $\text{PosUI}$ )

Fixed two posets  $\text{kdom}$  and  $\text{kcod}$  and consider the hom-set  $\text{PosLI}(\text{kdom}, \text{kcod})$ . Two generic morphisms in the homset have the form

$$\ell_1 : \text{kdom} \rightarrow_{\text{PosLI}} \text{kcod} \{\mathbf{I}_1\} \quad (49)$$

$$\ell_2 : \text{kdom} \rightarrow_{\text{PosLI}} \text{kcod} \{\mathbf{I}_2\} \quad (50)$$

and have in general different implementation posets  $\mathbf{I}_1$  and  $\mathbf{I}_2$ . We define an ordering by the projections of the morphisms:

$$\ell_1 \leq_{\text{PosLI}} \ell_2 \iff \text{proj}(\ell_1) \leq_{\text{PosL}} \text{proj}(\ell_2), \quad (51)$$

thus ignoring the implementation poset  $\mathbf{I}$ . Similarly, we can define an ordering on the hom-set  $\mathbf{PosUI}(\mathbf{kdom}, \mathbf{kcod})$ , by setting

$$u_1 \leq_{\mathbf{PosUI}} u_2 \iff \text{proj}(u_1) \leq_{\mathbf{PosU}} \text{proj}(u_2). \quad (52)$$

This relation is reflexive and transitive, but it is not antisymmetric, because there are different morphisms that have the same projection. Thus, the hom-sets  $\mathbf{PosLI}(\mathbf{F}, \mathbf{R})$  and  $\mathbf{PosUI}(\mathbf{F}, \mathbf{R})$  are pre-orders.

#### 14.4. DPI queries as $\mathbf{PosUI}/\mathbf{PosLI}$ morphisms

Having defined the categories  $\mathbf{PosLI}$  and  $\mathbf{PosUI}$ , we can now define the queries as  $\mathbf{PosLI}$  and  $\mathbf{PosUI}$  morphisms.

**Definition 14.17** (DPI queries)

Given a DPI  $\mathbf{d} : \mathbf{F} \rightarrow_{\mathbf{DPI}} \mathbf{R}$ , we can define the following queries:

$$\mathbf{FR} \mathbf{d} : \mathbf{F} \rightarrow_{\mathbf{PosL}} \mathbf{R} \quad (53)$$

$$\mathbf{RF} \mathbf{d} : \mathbf{R} \rightarrow_{\mathbf{PosU}} \mathbf{F} \quad (54)$$

$$\mathbf{FRI} \mathbf{d} : \mathbf{F} \rightarrow_{\mathbf{PosUI}} \mathbf{R}\{\mathbf{I}\} \quad (55)$$

$$\mathbf{RFI} \mathbf{d} : \mathbf{R} \rightarrow_{\mathbf{PosLI}} \mathbf{F}\{\mathbf{I}\} \quad (56)$$

$$\mathbf{FRB} \mathbf{d} : \mathbf{F} \rightarrow_{\mathbf{PosUI}} \mathbf{R}\{\mathcal{B}\} \quad (57)$$

$$\mathbf{RFB} \mathbf{d} : \mathbf{R} \rightarrow_{\mathbf{PosLI}} \mathbf{F}\{\mathcal{B}\} \quad (58)$$

First, we define  $\mathbf{FRI}$  and  $\mathbf{RFI}$  as follows:

$$\begin{aligned} \mathbf{FRI} \mathbf{d} : \mathbf{F} &\rightarrow_{\mathbf{PosUI}} \mathbf{R}\{\mathbf{I}\} \\ f^* &\longmapsto \{\langle r, i \rangle \text{ such that } (f^* \leq_{\mathbf{F}} \text{prov}(i)) \wedge (\text{req}(i) \leq_{\mathbf{R}} r) \wedge \text{avail}(i) \wedge \text{feas}(i)\} \end{aligned} \quad (59)$$

$$\begin{aligned} \mathbf{RFI} \mathbf{d} : \mathbf{R} &\rightarrow_{\mathbf{PosLI}} \mathbf{F}\{\mathbf{I}\} \\ r &\longmapsto \{\langle f, i \rangle \text{ such that } (f^* \leq_{\mathbf{F}} \text{prov}(i)) \wedge (\text{req}(i) \leq_{\mathbf{R}} r) \wedge \text{avail}(i) \wedge \text{feas}(i)\} \end{aligned} \quad (60)$$

From those, we define the others as projections:

$$\begin{aligned} \mathbf{FR} \mathbf{d} : \mathbf{F} &\rightarrow_{\mathbf{PosU}} \mathbf{R} \\ f^* &\longmapsto \{r \text{ for } \langle r, i \rangle \in [\mathbf{FRI} \mathbf{d}](f^*)\} \end{aligned} \quad (61)$$

$$\begin{aligned} \mathbf{RF} \mathbf{d} : \mathbf{R} &\rightarrow_{\mathbf{PosL}} \mathbf{F} \\ r &\longmapsto \{f \text{ for } \langle f, i \rangle \in [\mathbf{RFI} \mathbf{d}](r)\} \end{aligned} \quad (62)$$

$$\begin{aligned} \mathbf{FRB} \mathbf{d} : \mathbf{F} &\rightarrow_{\mathbf{PosUI}} \mathbf{R}\{\mathcal{B}\} \\ f^* &\longmapsto \uparrow \{\langle r, \text{IB}(i) \rangle \text{ for } \langle r, i \rangle \in [\mathbf{FRI} \mathbf{d}](f^*)\} \end{aligned} \quad (63)$$

$$\begin{aligned} \mathbf{RFB} \mathbf{d} : \mathbf{R} &\rightarrow_{\mathbf{PosLI}} \mathbf{F}\{\mathcal{B}\} \\ r &\longmapsto \downarrow \{\langle r, \text{IB}(i) \rangle \text{ for } \langle r, i \rangle \in [\mathbf{RFI} \mathbf{d}](r)\} \end{aligned} \quad (64)$$

**Lemma 14.18.** By construction,  $\mathbf{FR}$  and  $\mathbf{RF}$  can be recovered as projections of the other queries:

$$\text{proj}(\mathbf{FRI} \mathbf{d}) = \mathbf{FR} \mathbf{d} \quad (65)$$

$$\text{proj}(\mathbf{RFI} \mathbf{d}) = \mathbf{RF} \mathbf{d} \quad (66)$$

$$\text{proj}(\mathbf{FRB} \mathbf{d}) = \mathbf{FR} \mathbf{d} \quad (67)$$

$$\text{proj}(\mathbf{RFB} \mathbf{d}) = \mathbf{RF} \mathbf{d} \quad (68)$$

**Definition 14.19** (FR-congruence of DPIs)

Given two DPIs  $\mathbf{d}_1, \mathbf{d}_2 : \mathbf{F} \rightarrow_{\mathbf{DPI}} \mathbf{R}\{\star\}$  we say that they are *FR-congruent*, written  $\mathbf{d}_1 \cong_{\mathbf{FR}} \mathbf{d}_2$ , if

$$\mathbf{FR} \mathbf{d}_1 = \mathbf{FR} \mathbf{d}_2 \quad \text{and} \quad \mathbf{RF} \mathbf{d}_1 = \mathbf{RF} \mathbf{d}_2 \quad (69)$$

**Lemma 14.20.** FR-congruence is compositional. For all compatible  $\mathbf{d}_1, \mathbf{d}_2, \mathbf{d}_3, \mathbf{d}_4$ , we have:

$$\frac{\mathbf{d}_1 \cong_{\text{FR}} \mathbf{d}_2}{\mathbf{d}_3 \circ \mathbf{d}_1 \cong_{\text{FR}} \mathbf{d}_3 \circ \mathbf{d}_2} \quad \frac{\mathbf{d}_1 \cong_{\text{FR}} \mathbf{d}_2}{\mathbf{d}_1 \circ \mathbf{d}_4 \cong_{\text{FR}} \mathbf{d}_3 \circ \mathbf{d}_4} \quad (70)$$

**Definition 14.21** (B-congruence of DPIs)

Given two DPIs  $\mathbf{d}_1, \mathbf{d}_2 : \mathbf{F} \rightarrow_{\text{DPI}} \mathbf{R}\{\mathcal{B}\}$ , we say that they are *B-congruent*, written  $\mathbf{d}_1 \cong_{\text{B}} \mathbf{d}_2$ , if

$$\text{FRB } \mathbf{d}_1 = \text{FRB } \mathbf{d}_2 \quad \text{and} \quad \text{RFB } \mathbf{d}_1 = \text{RFB } \mathbf{d}_2 \quad (71)$$

**Lemma 14.22.** B-congruence is compositional. For all compatible  $\mathbf{d}_1, \mathbf{d}_2, \mathbf{d}_3, \mathbf{d}_4$ , we have:

$$\frac{\mathbf{d}_1 \cong_{\text{B}} \mathbf{d}_2}{\mathbf{d}_3 \circ \mathbf{d}_1 \cong_{\text{B}} \mathbf{d}_3 \circ \mathbf{d}_2} \quad \frac{\mathbf{d}_1 \cong_{\text{B}} \mathbf{d}_2}{\mathbf{d}_1 \circ \mathbf{d}_4 \cong_{\text{B}} \mathbf{d}_3 \circ \mathbf{d}_4} \quad (72)$$

**Lemma 14.23.** B-congruence implies FR-congruence:

$$\frac{\mathbf{d}_1 \cong_{\text{B}} \mathbf{d}_2}{\mathbf{d}_1 \cong_{\text{FR}} \mathbf{d}_2} \quad (73)$$

## 14.5. Free-forgetful adjunction between DP and DPI

**Definition 14.24** (Projection of a DPI to a DP)

Given a DPI  $\mathbf{d} : \mathbf{F} \rightarrow_{\text{DPI}} \mathbf{R}$ , we define the projected DP

$$\begin{aligned} \text{proj } \mathbf{d} : \mathbf{F} &\rightarrow_{\text{DP}} \mathbf{R} \\ \langle f^*, r \rangle &\mapsto \exists i \in \mathbf{I} : (f^* \leq \text{prov}(i)) \wedge (\text{req}(i) \leq r) \wedge (\text{avail}(i)) \wedge (\text{feas}(i)) \end{aligned} \quad (74)$$

**Definition 14.25** (Embedding of a DP to a DPI)

Given a DP  $\mathbf{d} : \mathbf{F} \rightarrow_{\text{DP}} \mathbf{R}$ , we define the embedded DPI

$$\text{embed } \mathbf{d} : \mathbf{F} \rightarrow_{\text{DPI}} \mathbf{R}\{\mathbf{1}\} \quad (75)$$

by the data

$$\mathbf{I} = \text{P\_C\_ProductSmash}(\llbracket \mathbf{F}, \mathbf{R}^{\text{op}} \rrbracket) \quad (76)$$

$$\text{prov} : [f \mid r^*] \mapsto f \quad (77)$$

$$\text{req} : [f \mid r^*] \mapsto r^* \quad (78)$$

$$\text{avail} : [f^* \mid r] \mapsto \mathbf{d}(f^*, r) \quad (79)$$

$$\text{feas} : [f \mid r^*] \mapsto \top \quad (80)$$

$$\text{IB} : [f \mid r^*] \mapsto * \quad (81)$$

**Lemma 14.26.**

$$\text{proj}(\text{embed } \mathbf{d}) = \mathbf{d} \quad (82)$$

**Lemma 14.27.**

$$\text{embed}(\mathbf{d}_1 \circ \mathbf{d}_2) \cong_{\text{B}} \text{embed}(\mathbf{d}_1) \circ \text{embed}(\mathbf{d}_2) \quad (83)$$

**Lemma 14.28.**

$$\text{proj}(\mathbf{dp}_1 \circ \mathbf{dp}_2) = \text{proj}(\mathbf{dp}_1) \circ \text{proj}(\mathbf{dp}_2) \quad (84)$$

## 15. Scalable computation for DPI

### 15.1. $\mathbf{SPosUI}$ and $\mathbf{SPosLI}$

Just like we generalized  $\mathbf{PosU}$  and  $\mathbf{PosL}$  to  $\mathbf{SPosU}$  and  $\mathbf{SPosL}$ , we can generalize  $\mathbf{PosUI}$  and  $\mathbf{PosLI}$  to  $\mathbf{SPosUI}$  and  $\mathbf{SPosLI}$ .

#### Definition 15.1

Given three posets  $\mathbf{kdom}$ ,  $\mathbf{kcod}$ , and  $\mathbf{kimp}$ , and a pair of posets  $S = \langle S^{\oplus}, S^{\ominus} \rangle$ , referred to as the “optimistic” and “pessimistic” “resolution” posets, a morphism in  $\mathbf{SPosUI}$

$$su : \{S\} \mathbf{kdom} \rightarrow_{\mathbf{SPosUI}} \mathbf{kcod} \{\mathbf{kimp}\} \quad (1)$$

is defined by giving two maps

$$su^{\oplus} : S^{\oplus \text{op}} \rightarrow_{\mathbf{Pos}} (\mathbf{kdom} \rightarrow_{\mathbf{PosUI}} \mathbf{kcod} \{\mathbf{kimp}\}) \quad (2)$$

$$su^{\ominus} : S^{\ominus} \rightarrow_{\mathbf{Pos}} (\mathbf{kdom} \rightarrow_{\mathbf{PosUI}} \mathbf{kcod} \{\mathbf{kimp}\}) \quad (3)$$

that satisfy the following condition:

$$\forall o \in S^{\oplus} : \forall p \in S^{\ominus} : su^{\oplus}(p) \leq su^{\ominus}(o) \quad (4)$$

#### Definition 15.2

Given three posets  $\mathbf{kdom}$ ,  $\mathbf{kcod}$ , and  $\mathbf{kimp}$ , and a pair of posets  $S = \langle S^{\oplus}, S^{\ominus} \rangle$ , referred to as the “optimistic” and “pessimistic” “resolution” posets, a morphism in  $\mathbf{SPosLI}$

$$sl : \{S\} \mathbf{kdom} \rightarrow_{\mathbf{SPosLI}} \mathbf{kcod} \{\mathbf{kimp}\} \quad (5)$$

is defined by giving two maps

$$sl^{\oplus} : S^{\oplus \text{op}} \rightarrow_{\mathbf{Pos}} (\mathbf{kdom} \rightarrow_{\mathbf{PosLI}} \mathbf{kcod} \{\mathbf{kimp}\}) \quad (6)$$

$$sl^{\ominus} : S^{\ominus} \rightarrow_{\mathbf{Pos}} (\mathbf{kdom} \rightarrow_{\mathbf{PosLI}} \mathbf{kcod} \{\mathbf{kimp}\}) \quad (7)$$

that satisfy the following condition:

$$\forall o \in S^{\oplus} : \forall p \in S^{\ominus} : sl^{\oplus}(p) \leq sl^{\ominus}(o) \quad (8)$$

### 15.2. Approximation of DPI queries

#### Definition 15.3 (Approximation classes for DPI queries)

Given a DPI  $\mathbf{d} : \mathbf{F} \rightarrow_{\mathbf{DPI}} \mathbf{R}$ , in analogy with Def. 13.5 and Def. 13.7, we can define the following sets:

- $(\mathbf{FR}_f^{\checkmark} \mathbf{d}), (\mathbf{FR}_f^{\star} \mathbf{d}), (\mathbf{FR}_f^{\oplus} \mathbf{d}), (\mathbf{FR}_f^{\ominus} \mathbf{d})$  as the approximation classes of  $\mathbf{FR} \mathbf{d}$ ,
- $(\mathbf{RF}_f^{\checkmark} \mathbf{d}), (\mathbf{RF}_f^{\star} \mathbf{d}), (\mathbf{RF}_f^{\oplus} \mathbf{d}), (\mathbf{RF}_f^{\ominus} \mathbf{d})$  as the approximation classes of  $\mathbf{RF} \mathbf{d}$ ,
- $(\mathbf{FRI}_f^{\checkmark} \mathbf{d}), (\mathbf{FRI}_f^{\star} \mathbf{d}), (\mathbf{FRI}_f^{\oplus} \mathbf{d}), (\mathbf{FRI}_f^{\ominus} \mathbf{d})$  as the approximation classes of  $\mathbf{FRI} \mathbf{d}$ ,
- $(\mathbf{RFI}_f^{\checkmark} \mathbf{d}), (\mathbf{RFI}_f^{\star} \mathbf{d}), (\mathbf{RFI}_f^{\oplus} \mathbf{d}), (\mathbf{RFI}_f^{\ominus} \mathbf{d})$  as the approximation classes of  $\mathbf{RFI} \mathbf{d}$ ,
- $(\mathbf{FRB}_f^{\checkmark} \mathbf{d}), (\mathbf{FRB}_f^{\star} \mathbf{d}), (\mathbf{FRB}_f^{\oplus} \mathbf{d}), (\mathbf{FRB}_f^{\ominus} \mathbf{d})$  as the approximation classes of  $\mathbf{FRB} \mathbf{d}$ ,
- $(\mathbf{RFB}_f^{\checkmark} \mathbf{d}), (\mathbf{RFB}_f^{\star} \mathbf{d}), (\mathbf{RFB}_f^{\oplus} \mathbf{d}), (\mathbf{RFB}_f^{\ominus} \mathbf{d})$  as the approximation classes of  $\mathbf{RFB} \mathbf{d}$ ,

where

- “ $\checkmark$ ” indicates the consistent approximations;
- “ $\oplus$ ” indicates the optimistic approximations;
- “ $\ominus$ ” indicates the pessimistic approximations;
- “ $\star$ ” indicates the exact approximations;

**Lemma 15.4** (Compositionality of queries approximations). For  $? \in \{\checkmark, \ominus, \odot, \star\}$ , the following holds:

$$\begin{array}{c}
 \frac{su_1 \in (FR_f^? \mathbf{d}_1) \quad su_2 \in (FR_f^? \mathbf{d}_2)}{su_1 \circ su_2 \in FR_f^?(\mathbf{d}_1 \circ \mathbf{d}_2)} \qquad \frac{sl_1 \in (RF_f^? \mathbf{d}_1) \quad sl_2 \in (RF_f^? \mathbf{d}_2)}{sl_2 \circ sl_1 \in RF_f^?(\mathbf{d}_1 \circ \mathbf{d}_2)} \\
 \\
 \frac{su_1 \in (FRI_f^? \mathbf{d}_1) \quad su_2 \in (FRI_f^? \mathbf{d}_2)}{su_1 \circ su_2 \in FRI_f^?(\mathbf{d}_1 \circ \mathbf{d}_2)} \qquad \frac{sl_1 \in (RFI_f^? \mathbf{d}_1) \quad sl_2 \in (RFI_f^? \mathbf{d}_2)}{sl_2 \circ sl_1 \in RFI_f^?(\mathbf{d}_1 \circ \mathbf{d}_2)} \\
 \\
 \frac{su_1 \in (FRB_f^? \mathbf{d}_1) \quad su_2 \in (FRB_f^? \mathbf{d}_2)}{su_1 \circ su_2 \in FRB_f^?(\mathbf{d}_1 \circ \mathbf{d}_2)} \qquad \frac{sl_1 \in (RFB_f^? \mathbf{d}_1) \quad sl_2 \in (RFB_f^? \mathbf{d}_2)}{sl_2 \circ sl_1 \in RFB_f^?(\mathbf{d}_1 \circ \mathbf{d}_2)}
 \end{array} \tag{9}$$

Note that the order of the morphisms is reversed for the backward queries ( $sl_2 \circ sl_1$  instead of  $su_1 \circ su_2$ ).

## 16. Numerical approximation

In this chapter we will discuss how to do “consistent” numerical computations that approximates computation on real numbers.

### 16.1. Approximation of DPs

Suppose have an ambient poset  $\mathbf{A}$  and a subset  $\mathbf{M} \subset \mathbf{A}$  that is used to “represent” the values of  $\mathbf{A}$ .

We have then an  $n$ -ary operation defined on  $\mathbf{A}$ , say  $\odot_{\mathbf{A}} : \mathbf{A}^n \rightarrow_{\text{Pos}} \mathbf{A}$ , and we need to approximate the results with computations only available on  $\mathbf{M}$ .

**Example 16.1.** The ambient poset  $\mathbf{A}$  could be the set of real numbers  $\overline{\mathbb{R}}$  and the subset  $\mathbf{M}$  could be the set of IEEE 754 floating point numbers (excluding NaN). The generic operation  $\odot_{\mathbf{A}}$  could be the addition or multiplication of two real numbers.

There is no way to decide what is a “good” approximation of an operation without some context. In this case, the context is given by the way the operation is used to create a DP.

Consider the case when we use  $\odot_{\mathbf{A}}$  to create a DP by lifting:

$$\begin{aligned} \text{DP\_LiftU } \odot_{\mathbf{A}} : \mathbf{A}^n & \rightarrow_{\text{DP}} \mathbf{A} \\ \langle \langle f_1, \dots, f_n \rangle, r \rangle & \mapsto \odot_{\mathbf{A}}(\langle f_1, \dots, f_n \rangle) \leq_{\mathbf{A}} r \end{aligned} \quad (1)$$

(Or the symmetric case  $\text{DP\_LiftL } \odot_{\mathbf{A}}$ ).

The question we pose is: is there a way to approximate the operation  $\odot_{\mathbf{A}}$  on  $\mathbf{A}$  with another operation on  $\mathbf{M}$

$$\tilde{\odot}_{\mathbf{M}} : \mathbf{M}^n \rightarrow_{\text{Pos}} \mathbf{M} \quad (2)$$

such that the approximated DP  $\text{DP\_LiftU } \tilde{\odot}_{\mathbf{M}}$  approximates the original DP  $\text{DP\_LiftU } \odot_{\mathbf{A}}$ ?

### 16.2. Approximation of operations in complete lattices

We show an answer to the question in the case when  $\mathbf{M}$  is a complete lattice.

#### 16.2.1. Upper and lower approximations of the original operation

For a complete lattice  $\mathbf{M}$ , we can define two maps that map elements of  $\mathbf{A}$  to the “next up” and “previous down” elements of  $\mathbf{M}$ :

$$\begin{aligned} \text{next}_{\mathbf{M}}^{\mathbf{A}} : \mathbf{A} & \rightarrow_{\text{Pos}} \mathbf{M} \\ a & \mapsto \bigwedge \{m \in \mathbf{M} \text{ such that } m \leq_{\mathbf{M}} a\} \end{aligned} \quad (3)$$

and

$$\begin{aligned} \text{prev}_{\mathbf{M}}^{\mathbf{A}} : \mathbf{A} & \rightarrow_{\text{Pos}} \mathbf{M} \\ a & \mapsto \bigvee \{m \in \mathbf{M} \text{ such that } a \leq_{\mathbf{M}} m\} \end{aligned} \quad (4)$$

Note that the meet and join exist because  $\mathbf{M}$  is a complete lattice.

Using these one can define an upper and lower approximation of  $\odot_{\mathbf{A}}$  as

$$\tilde{\odot}_{\mathbf{M}}^U = \odot_{\mathbf{A}} \circ \text{next}_{\mathbf{M}}^{\mathbf{A}} \quad (5)$$

$$\tilde{\odot}_{\mathbf{M}}^L = \odot_{\mathbf{A}} \circ \text{prev}_{\mathbf{M}}^{\mathbf{A}} \quad (6)$$

By construction, we have that, for all  $y \in \mathbf{M}^n$ ,

$$\tilde{\odot}_{\mathbf{M}}^L(y) \leq_{\mathbf{A}} \odot_{\mathbf{A}}(y) \leq_{\mathbf{A}} \tilde{\odot}_{\mathbf{M}}^U(y) \quad (7)$$

**Example 16.2** (Rounding modes of floating point numbers). When  $A = \overline{\mathbb{R}}$  and  $M$  is the finite set of IEEE754 floating-point numbers (excluding NaN), the maps  $\text{next}_M^A$  and  $\text{prev}_M^A$  coincide with directed rounding to the nearest representable value below and above, respectively. These are not available as explicit operations usable by a programmer; rather, a programmer can specify the *rounding mode* of the operation. The following table shows the different rounding modes:

<code>roundTiesToEven</code>	round-to-nearest, ties-to-even (default)
<code>roundTiesToAway</code>	round-to-nearest, ties-to-away
<code>roundTowardPositive</code>	round-toward $+\infty$ (upward)
<code>roundTowardNegative</code>	round-toward $-\infty$ (downward)
<code>roundTowardZero</code>	round-toward 0 (truncate)

The default rounding mode is “round-to-nearest, ties-to-even” (`roundTiesToEven`). If a number is not representable, the hardware rounds to the nearest representable value.

The rounding modes that are useful for us in this context are `roundTowardPositive` and `roundTowardNegative`. Using these, we obtain directly the upper and lower approximations of the original operation:

$$+_{\text{float}}^{\text{roundTowardPositive}} = +_{\mathbb{R}} ; \text{next}_{\text{float}}^{\overline{\mathbb{R}}} \quad (8)$$

$$+_{\text{float}}^{\text{roundTowardNegative}} = +_{\mathbb{R}} ; \text{prev}_{\text{float}}^{\overline{\mathbb{R}}} \quad (9)$$

### 16.2.2. Comparing the DPs

We now want to compare the generated DPs with the original DP, restricted to the subset  $M$ .

#### Definition 16.3

For a  $d : F \rightarrow_{\text{DP}} R$  and subposets  $P \subseteq F$  and  $Q \subseteq R$ , we define the restricted  $d$  as:

$$\begin{aligned} \text{restrict}(P, d, Q) : P &\rightarrow_{\text{DP}} Q \\ \langle f^*, r \rangle &\longmapsto d(f^*, r) \end{aligned} \quad (10)$$

**Lemma 16.4.** In the case of  $\text{DP\_LiftU}(\odot)$ , substituting  $\tilde{\odot}_M^U$  we obtain a pessimistic approximation, in the sense that

$$\text{DP\_LiftU } \tilde{\odot}_M^U \leq \text{restrict}(M^n, \text{DP\_LiftU } \odot, M) \quad (11)$$

*Proof.* Let’s test the feasibility of the DPs at the test point  $f^*, r$ . First, we note that the restriction is just a formal operation and does not change the feasibility of the DP:

$$[\text{restrict}(M^n, \text{DP\_LiftU } \odot, M)](f^*, r) = [\text{DP\_LiftU } \odot](f^*, r) = \odot_A(f^*) \leq_A r \quad (12)$$

For the other we see that

$$[\text{DP\_LiftU } \tilde{\odot}_M^U](f^*, r) = \tilde{\odot}_M^U(f^*) \leq_A r \quad (13)$$

Because of  $\odot_A(f^*) \leq \tilde{\odot}_M^U(f^*)$  we have that

$$\tilde{\odot}_M^U(f^*) \leq_A r \Rightarrow \odot_A(f^*) \leq_A r \quad (14)$$

Therefore, we have that

$$\text{DP\_LiftU } \tilde{\odot}_M^U \leq_{\text{DP}} \text{DP\_LiftU } \odot_A \quad (15)$$

□

Therefore, we have that a solution of  $\text{DP\_LiftU } \tilde{\odot}_M^U$  is a pessimistic solution of the original DP:

$$\text{FR}(\text{DP\_LiftU } \tilde{\odot}_M^U) \leq_{\text{PosU}} \text{FR}(\text{restrict}(M^n, \text{DP\_LiftU } \odot, M)) \quad (16)$$

$$\text{RF}(\text{DP\_LiftU } \tilde{\odot}_M^U) \leq_{\text{PosL}} \text{RF}(\text{restrict}(M^n, \text{DP\_LiftU } \odot, M)) \quad (17)$$

This is the symmetric result.

**Lemma 16.5.** In the case of  $\text{DP\_LiftL}(\odot)$ , substituting  $\tilde{\odot}_M^L$  we obtain a pessimistic approximation:

$$\text{DP\_LiftL } \tilde{\odot}_M^L \leq \text{restrict}(M, \text{DP\_LiftL } \odot, M^n) \quad (18)$$

Using the pessimistic approximation, we can make sure that any result we compute on the approximated poset is feasible in the original poset.



Part E.

Catalogs

## 17. Sets and posets catalog

### 17.1. Sets constructions

#### Definition 17.1 (Power set)

Given a poset  $\mathbf{A}$ , the *power set* is the poset  $\mathbf{S\_C\_Power}(\mathbf{A})$  whose elements are subsets of  $\mathbf{A}$ .

#### 17.1.1. Cartesian products

##### Definition 17.2 (Cartesian product)

Given a list of  $n$  sets  $\llbracket \mathbf{A}_k \rrbracket$ , the *Cartesian product* is the set  $\mathbf{S\_C\_Product}(\llbracket \mathbf{A}_k \rrbracket)$  with elements being tuples of elements of the sets.

##### Definition 17.3 (Smash product)

Given a list of  $n$  sets  $\llbracket \mathbf{A}_k \rrbracket$ , the *smash product* is the set  $\mathbf{S\_C\_ProductSmash}(\llbracket \mathbf{A}_k \rrbracket)$  where the elements are heterogeneous lists of elements from the sets.

The need for this construction arises from the fact that the Cartesian product of posets is not associative on the nose:  $\mathbf{P} \times (\mathbf{Q} \times \mathbf{R}) \neq (\mathbf{P} \times \mathbf{Q}) \times \mathbf{R}$ , but only up to isomorphism. In some cases, we want to have to have a product that is strictly associative.

We then work in a category where the objects are posets whose carrier sets are heterogenous tuples.

#### 17.1.2. Sum

##### Definition 17.4 (Sum)

Given a list of  $n$  sets  $\llbracket \mathbf{A}_k \rrbracket$ , the *sum* is the set  $\mathbf{S\_C\_Sum}(\llbracket \mathbf{A}_k \rrbracket)$  whose elements are pairs of an index  $k$  and an element  $x \in \mathbf{A}_k$ .

### 17.2. Constructions for single posets

#### 17.2.1. Opposite of a poset

##### Definition 17.5

The *opposite* of a poset  $\mathbf{P}$  is the poset  $\mathbf{P\_C\_Opposite}(\mathbf{P})$  with the same elements and the opposite order.

*This construction is described by the schema  $\mathbf{P\_C\_Opposite}$  (Section 26.2.11).*

#### 17.2.2. Arrow constructions

##### Definition 17.6 (Twisted arrow construction)

Given a poset  $\mathbf{P}$ , the *twisted arrow construction* is the poset  $\mathbf{P\_C\_Twisted}(\mathbf{P})$  with elements in  $\mathbf{S\_C\_Product}(\llbracket \mathbf{P}, \mathbf{P} \rrbracket)$  representing intervals ordered by inclusion:

$$\langle x_1, x_2 \rangle \leq_{\mathbf{TwP}} \langle y_1, y_2 \rangle \iff x_1 \leq_{\mathbf{P}} y_1 \text{ and } y_2 \leq_{\mathbf{P}} x_2.$$

*This construction is described by the schema  $\mathbf{P\_C\_Twisted}$  (Section 26.2.13).*

##### Definition 17.7 (Arrow construction)

Given a poset  $\mathbf{P}$ , the *arrow construction* of  $\mathbf{P}$  is the poset  $\mathbf{P\_C\_Arrow}(\mathbf{P})$  with elements in  $\mathbf{S\_C\_Product}(\llbracket \mathbf{P}, \mathbf{P} \rrbracket)$  and the order given by

$$\langle x_1, x_2 \rangle \leq \langle y_1, y_2 \rangle \iff x_1 \leq_{\mathbf{P}} y_1 \text{ and } x_2 \leq_{\mathbf{P}} y_2.$$

*This construction is described by the schema  $\mathbf{P\_C\_Arrow}$  (Section 26.2.8).*

### 17.2.3. Discretized version of a poset

**Definition 17.8** (Discretized version of a poset)

Given a poset  $P$ , the *discretized version* of  $P$  is the poset  $P\_C\_Discretized(P)$  with elements in  $P$  and the order given by equality.

*This construction is described by the schema  $P\_C\_Discretized$  (Section 26.2.9).*

### 17.2.4. Posets of subsets

**Definition 17.9**

Given a poset  $P$ , we define the poset  $P\_C\_Power(P)$  whose carrier is  $S\_C\_Power(P)$  and order is given by set inclusion.

*This construction is described by the schema  $P\_C\_Power$  (Section 26.2.12).*

**Definition 17.10**

Given a poset  $P$ , we define the poset  $P\_C\_LowerSets(P)$ , also indicated as  $L P$ , whose carrier is the subset of  $S\_C\_Power(P)$  of lower sets, and order is given by set inclusion.

*This construction is described by the schema  $P\_C\_LowerSets$  (Section 26.2.10).*

**Definition 17.11**

Given a poset  $P$ , we define the poset  $P\_C\_UpperSets(P)$ , also indicated as  $U P$ , whose carrier is the subset of  $S\_C\_Power(P)$  of upper sets, and order is given by set inclusion.

*This construction is described by the schema  $P\_C\_UpperSets$  (Section 26.2.15).*

## 17.3. Constructions with multiple posets

### 17.3.1. Cartesian product of posets

**Definition 17.12**

Given a list of  $n$  posets  $\llbracket P_k \rrbracket$ , the *Cartesian product* is the poset  $P\_C\_Product(\llbracket P_k \rrbracket)$  with elements in  $S\_C\_Product(\llbracket P_k \rrbracket)$  and the order given by

$$\langle p_1, \dots, p_n \rangle \leq \langle p'_1, \dots, p'_n \rangle \iff p_k \leq_{P_k} p'_k \text{ for all } i \in \{1, \dots, n\}.$$

*This construction is described by the schema  $P\_C\_Product$  (Section 26.2.17).*

**Lemma 17.13.** If the posets  $\llbracket P_k \rrbracket$  are dcpo (fcpo), then  $P\_C\_Product(\llbracket P_k \rrbracket)$  is dcpo (fcpo).

**Smash product of posets** The smash product  $P\_C\_ProductSmash(\llbracket P_k \rrbracket)$  of a list of posets is a poset isomorphic to the Cartesian product of the posets, but where elements are represented as “exploded” tuples, using the  $S\_C\_ProductSmash$  constructor.

*This construction is described by the schema  $P\_C\_ProductSmash$  (Section 26.2.18).*

### 17.3.2. Direct sum of posets

**Definition 17.14**

Given a list of  $n$  posets  $\llbracket P_k \rrbracket$ , the *direct sum* is the poset  $P\_C\_Sum(\llbracket P_k \rrbracket)$  with elements in  $S\_C\_Sum(\llbracket P_k \rrbracket)$  and the order given by

$$\langle kx \rangle \leq \langle j, y \rangle \iff k = j \text{ and } x \leq_{P_k} y.$$

*This construction is described by the schema  $P\_C\_Sum$  (Section 26.2.19).*

**Lemma 17.15.** If the posets  $\llbracket P_k \rrbracket$  are dcpo (fcpo), then  $P\_C\_Sum(\llbracket P_k \rrbracket)$  is dcpo (fcpo).

**Direct (smash) sum of posets** The smash sum of a list of posets  $P\_C\_SumSmash(\llbracket P_k \rrbracket)$  is a poset isomorphic to the direct sum of the posets. It is mathematically equivalent to the direct sum, but the carrier set is a heterogenous tuple.

*This construction is described by the schema  $P\_C\_SumSmash$  (Section 26.2.20).*

### 17.3.3. Lexicographic product of posets

#### Definition 17.16

Given a list of  $n$  posets  $\llbracket P_k \rrbracket$ , the *lexicographic product* is the poset  $P\_C\_Lexicographic(\llbracket P_k \rrbracket)$  with elements in  $S\_C\_Product(\llbracket P_k \rrbracket)$  and the order given by

$$\langle p_1, \dots, p_n \rangle \leq \langle p'_1, \dots, p'_n \rangle \iff \exists k \in \{1, \dots, n\} : p_i = p'_i \text{ for all } i < k \text{ and } p_k \leq_{P_k} p'_k.$$

*This construction is described by the schema  $P\_C\_Lexicographic$  (Section 26.2.16).*

## 17.4. Poset Filters

These constructions are “filters”: they describe a subposet of a poset.

### 17.4.1. Finite subposet of an ambient poset

This is the most basic filter. Given a finite subset  $A \subseteq P$  of a poset  $P$ , we call  $P\_F\_Subposet(P, A)$  the subposet generated by that subset.

*This construction is described by the schema  $P\_F\_Subposet$  (Section 26.2.26).*

### 17.4.2. Interval in a poset

#### Definition 17.17

Given a poset  $P$  and elements  $x, y \in P$ , the *interval* between  $x$  and  $y$  is the subposet  $P\_F\_Interval(P, x, y)$  of  $P$  which contains the elements  $p \in P$  such that  $x \leq_P p \leq_P y$ .

*This construction is described by the schema  $P\_F\_Interval$  (Section 26.2.24).*

### 17.4.3. Lower and upper closure in a poset

#### Definition 17.18

Given a poset  $P$  and a subset  $A \subseteq P$ , the *upper closure* of  $A$  in  $P$  is the subposet  $P\_F\_UpperClosure(P, A)$  whose elements are in  $\uparrow A$ .

*This construction is described by the schema  $P\_F\_UpperClosure$  (Section 26.2.27).*

#### Definition 17.19

Given a poset  $P$  and a subset  $A \subseteq P$ , the *lower closure* of  $A$  in  $P$  is the subposet  $P\_F\_LowerClosure(P, A)$  whose elements are in  $\downarrow A$ .

*This construction is described by the schema  $P\_F\_LowerClosure$  (Section 26.2.25).*

As a rule, the set  $A$  is taken to be an antichain ( $A = \text{Min } A$ ).

#### 17.4.4. Union and Intersection of sub posets

Consider an ambient poset  $P$  and a list of subposets  $\llbracket Q_k \rrbracket$  such that each is a subposet of  $P$ :  $Q_k \subseteq P$ .

Because they are all subposets of  $P$ , they have the same type of elements, and they have the same order; they just differ in the carrier set.

We can then consider the union  $P\_F\_Union(\llbracket Q_k \rrbracket)$  and intersection  $P\_F\_Intersection(\llbracket Q_k \rrbracket)$  of these carrier sets.

*This construction is described by the schema  $P\_F\_C\_Intersection$  (Section 26.2.22).*

*This construction is described by the schema  $P\_F\_C\_Union$  (Section 26.2.23).*

#### 17.4.5. Sampling a poset

If the poset  $P$  is numeric, contains a copy of the integers, and has defined the operations of sum  $+_P$  and multiplication  $\cdot_P$ , we can identify a subposet by “sampling” the poset.

Given an “offset”  $O$  and a “step”  $S$  we can define the subset

$$A = \{O +_P i \cdot_P S \text{ for } i \in \mathbb{Z}\} \quad (1)$$

For example, given the poset  $P = \mathbb{R}$ , we can define the poset of odd integers by setting  $O = 1$  and  $S = 2$ .

More in general, it is useful to generalize the construction and require 5 values that satisfy:

$$B \leq_P L \leq_P O \leq_P H \leq_P T \quad (2)$$

and the subposet is defined by

$$A = \{B, T\} \cup ([L, H] \cap \{O + i \cdot S \text{ for } i \in \mathbb{Z}\}) \quad (3)$$

We call this construction  $P\_F\_Bounded(P, B, L, O, H, T)$ .

Having this way of parametrizing a subposet is useful because it allows to do *abstract interpretation* of numerical operations.

## 18. Monotone maps catalog

In the previous chapter we have explained why it is important to characterize whether a map  $f$  is Scott co-continuous or Scott continuous and the role that the upper preimage  $\mathbf{Ui}f$  and the lower preimage  $\mathbf{Li}f$  play in computing the solution of DPs.

In this chapter we present a catalog of monotone maps that can be used as building blocks for constructing DPs.

### 18.1. Identity map

We collect here some properties of common functions that are useful to know.

**Definition 18.1** (Identity map)

Given a poset  $P$ , the identity map is the map

$$\begin{aligned} \mathbf{M\_Id}(P) : P &\rightarrow_{\text{Pos}} P \\ x &\longmapsto x \end{aligned} \tag{1}$$

*This construction is described by the schema  $\mathbf{M\_Id}$  (Section 26.3.4).*

**Lemma 18.2** (Identity). If  $P$  is a dcpo (resp. fcpo), the identity map  $\text{id}_P : P \rightarrow P$  is Scott continuous (resp. Scott co-continuous).

*Proof.* Both properties are immediate: the identity preserves *all* (co)limits.  $\square$

### 18.2. Constant maps

**Definition 18.3** (Constant map)

Given two posets  $P$  and  $Q$  and an element  $c \in Q$ , the constant map is the map

$$\begin{aligned} \mathbf{M\_Constant}(P, c) : P &\rightarrow_{\text{Pos}} Q \\ x &\longmapsto c \end{aligned} \tag{2}$$

*This construction is described by the schema  $\mathbf{M\_Constant}$  (Section 26.3.1).*

**Lemma 18.4** (Constant maps). Let  $P$  be a dcpo and  $Q$  any poset. Every constant map from  $P$  to  $Q$  is Scott continuous. Dually, if  $P$  is an fcpo, every constant map from  $P$  to  $Q$  is Scott co-continuous.

### 18.3. Ceiling and floor

**Lemma 18.5.** The function  $\text{ceil}$  is Scott continuous and we have that

$$\mathbf{Ui} \text{ceil} = \uparrow \text{floor} \tag{3}$$

$$\mathbf{Li} \text{ceil} = \downarrow \text{floor} \tag{4}$$

**Lemma 18.6.** The function  $\text{floor}$  is Scott co-continuous and we have that

$$\mathbf{Ui} \text{floor} = \uparrow \text{ceil} \tag{5}$$

$$\mathbf{Li} \text{floor} = \downarrow \text{ceil} \tag{6}$$

#### 18.3.1. Generalized rounding

Based on  $\text{ceil}$  we define a family of maps parameterized by a step  $S$  and an offset  $O$ .

**Definition 18.7**

Given a step  $S > 0$  and an offset  $O$ , we define the map

$$\text{M\_RoundUp}(S, O) : x \mapsto \text{ceil}((x - O)/S) \cdot S + O \quad (7)$$

This construction is described by the schema [M\\_RoundUp](#) (Section 26.3.47).

**Definition 18.8**

Given a step  $S > 0$  and an offset  $O$ , we define the map

$$\text{M\_RoundDown}(S, O) : x \mapsto \text{floor}((x - O)/S) \cdot S + O \quad (8)$$

This construction is described by the schema [M\\_RoundDown](#) (Section 26.3.46).

We recover the ceiling and floor functions as special cases:

$$\text{M\_RoundUp}(1, 0) = \text{ceil} \quad (9)$$

$$\text{M\_RoundDown}(1, 0) = \text{floor} \quad (10)$$

**Lemma 18.9.**

$$\text{Ui}(\text{M\_RoundUp}(S, O)) = \uparrow \text{M\_RoundDown}(S, O) \quad (11)$$

$$\text{Li}(\text{M\_RoundUp}(S, O)) = \downarrow \text{M\_RoundDown}(S, O) \quad (12)$$

$$\text{Ui}(\text{M\_RoundDown}(S, O)) = \uparrow \text{M\_RoundUp}(S, O) \quad (13)$$

$$\text{Li}(\text{M\_RoundDown}(S, O)) = \downarrow \text{M\_RoundUp}(S, O) \quad (14)$$

*Proof.*

Let  $\text{mul}_S$  be the map  $\text{mul}_S : x \mapsto x \cdot S$ . Then  $(\text{mul}_S)^{-1} = \text{mul}_{1/S}$ ,  $\text{Ui}(\text{mul}_S) = \uparrow \text{mul}_{1/S}$  and  $\text{Li}(\text{mul}_S) = \downarrow \text{mul}_{1/S}$ .

Let  $\text{add}_O$  be the map  $\text{add}_O : x \mapsto x + O$ . Then  $(\text{add}_O)^{-1} = \text{add}_{-O}$ ,  $\text{Ui}(\text{add}_O) = \uparrow \text{add}_{-O}$  and  $\text{Li}(\text{add}_O) = \downarrow \text{add}_{-O}$ .

Now write  $\text{M\_RoundUp}(S, O)$  as a composition of  $\text{mul}_S$  and  $\text{add}_O$  as follows:

$$\text{M\_RoundUp}(S, O) = \text{add}_{-O} \circ \text{mul}_{1/S} \circ \text{ceil} \circ \text{mul}_S \circ \text{add}_O \quad (15)$$

And likewise for  $\text{M\_RoundDown}(S, O)$  we have that

$$\text{M\_RoundDown}(S, O) = \text{add}_{-O} \circ \text{mul}_{1/S} \circ \text{floor} \circ \text{mul}_S \circ \text{add}_O \quad (16)$$

And now just compute the upper and lower inverses for  $\text{M\_RoundUp}(S, O)$ :

$$\text{Ui}(\text{M\_RoundUp}(S, O)) = \text{Ui}(\text{add}_{-O} \circ \text{mul}_{1/S} \circ \text{ceil} \circ \text{mul}_S \circ \text{add}_O) \quad (17)$$

$$= \text{Ui}(\text{add}_O) \circ \text{Ui}(\text{mul}_S) \circ \text{Ui}(\text{ceil}) \circ \text{Ui}(\text{mul}_{1/S}) \circ \text{Ui}(\text{add}_{-O}) \quad (18)$$

$$= \uparrow \text{add}_{-O} \circ \uparrow \text{mul}_{1/S} \circ \uparrow \text{floor} \circ \uparrow \text{mul}_S \circ \uparrow \text{add}_O \quad (19)$$

$$= \uparrow (\text{add}_{-O} \circ \text{mul}_{1/S} \circ \text{floor} \circ \text{mul}_S \circ \text{add}_O) \quad (20)$$

$$= \uparrow \text{M\_RoundDown}(S, O) \quad (21)$$

$$\text{Li}(\text{M\_RoundUp}(S, O)) = \text{Li}(\text{add}_{-O} \circ \text{mul}_{1/S} \circ \text{ceil} \circ \text{mul}_S \circ \text{add}_O) \quad (22)$$

$$= \text{Li}(\text{add}_O) \circ \text{Li}(\text{mul}_S) \circ \text{Li}(\text{ceil}) \circ \text{Li}(\text{mul}_{1/S}) \circ \text{Li}(\text{add}_{-O}) \quad (23)$$

$$= \downarrow \text{add}_{-O} \circ \downarrow \text{mul}_{1/S} \circ \downarrow \text{floor} \circ \downarrow \text{mul}_S \circ \downarrow \text{add}_O \quad (24)$$

$$= \downarrow (\text{add}_{-O} \circ \text{mul}_{1/S} \circ \text{floor} \circ \text{mul}_S \circ \text{add}_O) \quad (25)$$

$$= \downarrow \text{M\_RoundDown}(S, O) \quad (26)$$

Similarly for  $\text{M\_RoundDown}(S, O)$ :

$$\text{Ui}(\text{M\_RoundDown}(S, O)) = \text{Ui}(\text{add}_{-O} \circ \text{mul}_{1/S} \circ \text{floor} \circ \text{mul}_S \circ \text{add}_O) \quad (27)$$

$$= \text{Ui}(\text{add}_O) \circ \text{Ui}(\text{mul}_S) \circ \text{Ui}(\text{floor}) \circ \text{Ui}(\text{mul}_{1/S}) \circ \text{Ui}(\text{add}_{-O}) \quad (28)$$

$$= \uparrow \text{add}_{-O} \circ \uparrow \text{mul}_{1/S} \circ \uparrow \text{ceil} \circ \uparrow \text{mul}_S \circ \uparrow \text{add}_O \quad (29)$$

$$= \uparrow (\text{add}_{-O} \circ \text{mul}_{1/S} \circ \text{ceil} \circ \text{mul}_S \circ \text{add}_O) \quad (30)$$

$$= \uparrow \text{M\_RoundUp}(S, O) \quad (31)$$

$$\text{Li}(\text{M\_RoundDown}(S, O)) = \text{Li}(\text{add}_{-O} \circ \text{mul}_{1/S} \circ \text{floor} \circ \text{mul}_S \circ \text{add}_O) \quad (32)$$

$$= \text{Li}(\text{add}_O) \circ \text{Li}(\text{mul}_S) \circ \text{Li}(\text{floor}) \circ \text{Li}(\text{mul}_{1/S}) \circ \text{Li}(\text{add}_{-O}) \quad (33)$$

$$= \downarrow \text{add}_{-O} \circ \downarrow \text{mul}_{1/S} \circ \downarrow \text{ceil} \circ \downarrow \text{mul}_S \circ \downarrow \text{add}_O \quad (34)$$

$$= \downarrow (\text{add}_{-O} \circ \text{mul}_{1/S} \circ \text{ceil} \circ \text{mul}_S \circ \text{add}_O) \quad (35)$$

$$= \downarrow \text{M\_RoundUp}(S, O) \quad (36)$$

□

## 18.4. Sum, multiplication, and division

In this section we consider the sum ([add](#)), multiplication ([mul](#)), and division ([div](#)) of real numbers, and we study their extensions to the poset completions (e.g. adding a  $+\infty$  and a  $-\infty$  element to  $\mathbb{R}$ ). We find that for each of those there are two distinct extensions: one that is Scott continuous and one that is Scott co-continuous, and the two agree everywhere except at singularity.

For example, for addition,  $\text{add}(+\infty, -\infty)$  is undefined, but  $\text{add}\uparrow(+\infty, -\infty) = -\infty$  and  $\text{add}\downarrow(+\infty, -\infty) = +\infty$ .

For division we find that while  $\text{div}(0, 0)$  is undefined on the reals, we have that  $\text{div}\uparrow(0, 0) = \text{div}\uparrow(+\infty, +\infty) = 0$  and  $\text{div}\downarrow(0, 0) = \text{div}\downarrow(+\infty, +\infty) = +\infty$ .

### 18.4.1. Sum

We consider addition on real numbers

$$\text{add} : \mathbb{R} \times \mathbb{R} \rightarrow_{\text{Pos}} \mathbb{R} \quad (37)$$

and we ask about its Scott-continuity properties.

First of all, we need to make  $\mathbb{R}$  a dcpo and fcpo by adding the elements  $+\infty$  and  $-\infty$ . We call this completion  $\overline{\mathbb{R}}$ .

**Definition 18.10** (Completion of real numbers)

The completion of real numbers  $\mathbb{R}$  is the set  $\overline{\mathbb{R}} = \mathbb{R} \cup \{+\infty, -\infty\}$  with the order  $\leq$  extended in the obvious way.

We now need to extend the addition operation to  $\overline{\mathbb{R}} \times \overline{\mathbb{R}}$ .

Some choices are obvious. For a finite real number  $x$ , we define  $\text{add}(x, +\infty) = +\infty$  and  $\text{add}(x, -\infty) = -\infty$ . And we set  $\text{add}(+\infty, +\infty) = +\infty$  and  $\text{add}(-\infty, -\infty) = -\infty$ .

What about the case  $\text{add}(-\infty, +\infty)$ ? Should it be  $+\infty$  or  $-\infty$ ?

It turns out that depending on the choice of  $\text{add}(-\infty, +\infty)$  we obtain a Scott-continuous or a co-Scott-continuous function.

**Lemma 18.11.** The Scott continuous extension  $\text{add}\uparrow : \overline{\mathbb{R}} \times \overline{\mathbb{R}} \rightarrow_{\text{Pos}} \overline{\mathbb{R}}$  to the addition operation is given by the following rules (where  $x, y \in \mathbb{R}$ ):

$$\text{add}\uparrow(x, y) = \text{add}(x, y) \quad (38)$$

$$\text{add}\uparrow(-\infty, y) = -\infty \quad (39)$$

$$\text{add}\uparrow(+\infty, y) = +\infty \quad (40)$$

$$\text{add}\uparrow(-\infty, +\infty) = -\infty \quad (41)$$

*Proof.* If we want that  $\text{add}\uparrow$  is Scott continuous, we need to have that for every directed  $\mathbf{D} \subseteq \overline{\mathbb{R}} \times \overline{\mathbb{R}}$  we have that

$$\text{add}\uparrow\left(\bigvee^{\uparrow} \mathbf{D}\right) = \bigvee^{\uparrow} \{\text{add}\uparrow(\langle x, y \rangle) \mid \langle x, y \rangle \in \mathbf{D}\} \quad (42)$$

We now carefully choose a directed set  $\mathbf{D}$  to obtain the value of  $\text{add}\uparrow(\langle -\infty, +\infty \rangle)$ . Take

$$\mathbf{D} = \{\langle -\infty, i \rangle \mid i \in \mathbb{N}\} \quad (43)$$



This set is directed and  $\bigvee^\uparrow \mathbf{D} = \langle -\infty, +\infty \rangle$ . Then (42) becomes

$$\text{add}\uparrow(-\infty, +\infty) = \bigvee^\uparrow \{\text{add}\uparrow(-\infty, i) \mid i \in \mathbb{N}\} = \bigvee^\uparrow \{-\infty\} = -\infty \quad (44)$$

□

**Lemma 18.12** (Lower pre-image of  $\text{add}\uparrow$ ). For  $q \in \overline{\mathbb{R}}$  we have

$$\text{Li add}\uparrow : q \mapsto \begin{cases} \downarrow\{\langle -\infty, +\infty \rangle, \langle +\infty, -\infty \rangle\}, & \text{if } q = -\infty, \\ \downarrow\{\langle -\infty, +\infty \rangle, \langle +\infty, -\infty \rangle\} \cup \downarrow\{(x, y) \in \mathbb{R}^2 : x + y = q\}, & \text{if } q \in \mathbb{R}, \\ \downarrow\langle +\infty, +\infty \rangle, & \text{if } q = +\infty. \end{cases}$$

**Lemma 18.13** (Upper pre-image of  $\text{add}\uparrow$ ). For  $p \in \overline{\mathbb{R}}$ :

$$\text{Ui add}\uparrow : p \mapsto \begin{cases} \uparrow\langle -\infty, -\infty \rangle, & \text{if } p = -\infty, \\ (\{+\infty\} \times \uparrow\langle -\infty \rangle) \cup ((\uparrow\langle -\infty \rangle \times \{+\infty\}) \cup \uparrow\{(x, y) \in \mathbb{R}^2 : x + y = p\}), & \text{if } p \in \mathbb{R}, \\ (\{+\infty\} \times \uparrow\langle -\infty \rangle) \cup ((\uparrow\langle -\infty \rangle \times \{+\infty\}), & \text{if } p = +\infty. \end{cases}$$

Note that in the cases  $p = +\infty$  and  $p \in \mathbb{R}$  the upset cannot be written as the up closure of an antichain. We expect this because  $\text{add}\uparrow$  is not Scott co-continuous.

**Lemma 18.14.** The Scott co-continuous extension  $\text{add}\downarrow : \overline{\mathbb{R}} \times \overline{\mathbb{R}} \rightarrow_{\text{Pos}} \overline{\mathbb{R}}$  to the addition operation is given by the following rules (where  $x, y \in \mathbb{R}$ ):

$$\text{add}\downarrow(x, y) = \text{add}(x, y) \quad (45)$$

$$\text{add}\downarrow(-\infty, y) = -\infty \quad (46)$$

$$\text{add}\downarrow(+\infty, y) = +\infty \quad (47)$$

$$\text{add}\downarrow(-\infty, +\infty) = +\infty \quad (48)$$

*Proof.* For the Scott co-continuous extension  $\text{add}\downarrow$ , we choose the set

$$\mathbf{F} = \{\langle -i, +\infty \rangle \mid i \in \mathbb{N}\} \quad (49)$$

This set is filtered and  $\bigwedge^\downarrow \mathbf{F} = \langle -\infty, +\infty \rangle$ . Then (42) becomes

$$\text{add}\downarrow(-\infty, +\infty) = \bigwedge^\downarrow \{\text{add}\downarrow(-i, +\infty) \mid i \in \mathbb{N}\} = \bigwedge^\downarrow \{+\infty\} = +\infty \quad (50)$$

□

**Lemma 18.15** (Upper pre-image of  $\text{add}\downarrow$ ). For  $p \in \overline{\mathbb{R}}$ :

$$\text{Ui add}\downarrow : p \mapsto \begin{cases} \uparrow\langle -\infty, -\infty \rangle, & \text{if } p = -\infty, \\ \uparrow\{\langle -\infty, +\infty \rangle, \langle +\infty, -\infty \rangle\} \cup \uparrow\{(x, y) \in \mathbb{R}^2 : x + y = p\}, & \text{if } p \in \mathbb{R}, \\ \uparrow\{\langle -\infty, +\infty \rangle, \langle +\infty, -\infty \rangle\}, & \text{if } p = +\infty. \end{cases}$$

**Lemma 18.16** (Lower pre-image of  $\text{add}\downarrow$ ). For  $q \in \overline{\mathbb{R}}$ :

$$\text{Li add}\downarrow : q \mapsto \begin{cases} (\{-\infty\} \times \downarrow\{+\infty\}) \cup ((\downarrow\{+\infty\} \times \{-\infty\}), & \text{if } q = -\infty, \\ (\{-\infty\} \times \downarrow\{+\infty\}) \cup ((\downarrow\{+\infty\} \times \{-\infty\}) \cup \downarrow\{(x, y) \in \mathbb{R}^2 : x + y = q\}), & \text{if } q \in \mathbb{R}, \\ \downarrow\langle +\infty, +\infty \rangle, & \text{if } q = +\infty. \end{cases}$$

Note that in the cases  $q = -\infty$  and  $q \in \mathbb{R}$  the downset cannot be written as the down closure of an antichain. We expect this because  $\text{add}\downarrow$  is not Scott co-continuous.

## 18.4.2. Multiplication

We consider multiplication on the non-negative real numbers

$$\text{mul} : \mathbb{R}_{\geq 0} \times \mathbb{R}_{\geq 0} \rightarrow_{\text{Pos}} \mathbb{R}_{\geq 0} \quad (51)$$

and we ask about its Scott-(co)continuity properties.

First of all, we need to make  $\mathbb{R}_{\geq 0}$  a dcpo and an fcpo by adding the element  $+\infty$ . We call this completion  $\overline{\mathbb{R}}_{\geq 0}$ .

**Definition 18.17** (Completion of non-negative real numbers)

The completion of the non-negative real numbers  $\mathbb{R}_{\geq 0}$  is the set

$$\overline{\mathbb{R}}_{\geq 0} = \mathbb{R}_{\geq 0} \cup \{+\infty\},$$

ordered in the obvious way, with  $+\infty$  the top element.

We now need to extend the multiplication operation to  $\overline{\mathbb{R}}_{\geq 0} \times \overline{\mathbb{R}}_{\geq 0}$ .

Some choices are obvious. For a finite real number  $x \geq 0$ , we define  $\text{mul}(x, +\infty) = +\infty$  when  $x > 0$ , and  $\text{mul}(0, y) = 0$  for  $y \in \mathbb{R}_{\geq 0}$ . We also set  $\text{mul}(+\infty, +\infty) = +\infty$ .

What about the case  $\text{mul}(0, +\infty)$ , and symmetrically  $\text{mul}(+\infty, 0)$ ? Should it be  $+\infty$  or 0?

It turns out that depending on the choice of  $\text{mul}(0, +\infty)$  we obtain a Scott continuous or a Scott continuous function.

**Lemma 18.18.** The Scott continuous extension  $\text{mul}\uparrow : \overline{\mathbb{R}}_{\geq 0} \times \overline{\mathbb{R}}_{\geq 0} \rightarrow_{\text{Pos}} \overline{\mathbb{R}}_{\geq 0}$  to the multiplication operation is given by the following rules (where  $x, y \in \mathbb{R}_{\geq 0}$ ):

$$\text{mul}\uparrow(x, y) = \text{mul}(x, y) \quad (52)$$

$$\text{mul}\uparrow(0, y) = 0, \quad (53)$$

$$\text{mul}\uparrow(+\infty, y) = +\infty, \quad (54)$$

$$\text{mul}\uparrow(0, +\infty) = 0. \quad (55)$$

This construction is described by the schema **M\_MultiplyU** (Section 26.3.42).

*Proof.* The proof parallels the one for addition. For the Scott continuous extension, take the directed set

$$\mathbf{D} = \{\langle 0, n \rangle \mid n \in \mathbb{N}\},$$

whose supremum is  $\langle 0, +\infty \rangle$ . The Scott-continuity requirement (analogous to (42)) gives

$$\text{mul}\uparrow(0, +\infty) = \bigvee_{n \in \mathbb{N}}^{\uparrow} \text{mul}\uparrow(0, n) = \bigvee^{\uparrow} \{0\} = 0.$$

□

**Lemma 18.19** (Upper pre-image of  $\text{mul}\uparrow$ ). For  $r \in \overline{\mathbb{R}}_{\geq 0}$ :

$$\mathbf{Ui} \text{mul}\uparrow : r \mapsto \begin{cases} \uparrow\langle 0, 0 \rangle, & \text{if } r = 0, \\ (\{+\infty\} \times (\uparrow 0)) \cup ((\uparrow 0) \times \{+\infty\}) \cup \uparrow\{ \langle x, y \rangle \in \mathbb{R}_{\geq 0}^2 : \text{mul}(x, y) = r \}, & \text{if } 0 < r < +\infty, \\ (\{+\infty\} \times (\uparrow 0)) \cup ((\uparrow 0) \times \{+\infty\}), & \text{if } r = +\infty. \end{cases}$$

Note that in the case  $r = +\infty$  the upset cannot be written as the up closure of an antichain. We expect this because  $\text{mul}\uparrow$  is not Scott co-continuous.

**Lemma 18.20** (Lower pre-image of  $\text{mul}\uparrow$ ). For  $s \in \overline{\mathbb{R}}_{\geq 0}$ :

$$\mathbf{Li} \text{mul}\uparrow : s \mapsto \begin{cases} \downarrow\{ \langle 0, +\infty \rangle, \langle +\infty, 0 \rangle \}, & \text{if } s = 0, \\ \downarrow\{ \langle 0, +\infty \rangle, \langle +\infty, 0 \rangle \} \cup \downarrow\{ \langle x, y \rangle \in \mathbb{R}_{\geq 0}^2 \text{ such that } \text{mul}(x, y) = s \}, & \text{if } 0 < s < +\infty, \\ \downarrow\langle +\infty, +\infty \rangle, & \text{if } s = +\infty. \end{cases}$$

**Lemma 18.21.** The Scott co-continuous extension  $\text{mul}\downarrow : \overline{\mathbb{R}}_{\geq 0} \times \overline{\mathbb{R}}_{\geq 0} \rightarrow_{\text{Pos}} \overline{\mathbb{R}}_{\geq 0}$  to the multiplication operation is given by the following rules (where  $x, y \in \mathbb{R}_{\geq 0}$ ):

$$\text{mul}\downarrow(x, y) = \text{mul}(x, y) \quad (56)$$

$$\text{mul}\downarrow(0, y) = 0, \quad (57)$$

$$\text{mul}\downarrow(+\infty, y) = +\infty, \quad (58)$$

$$\text{mul}\downarrow(0, +\infty) = +\infty. \quad (59)$$

*Proof.* For the Scott co-continuous extension consider the filtered set

$$\mathbf{F} = \left\{ \left\langle \frac{1}{n+1}, +\infty \right\rangle \text{ for } n \in \mathbb{N} \right\},$$

whose infimum is again  $\langle 0, +\infty \rangle$ . Enforcing Scott-co-continuity yields

$$\text{mul}\downarrow(0, +\infty) = \bigwedge_{n \in \mathbb{N}} \text{mul}\downarrow\left(\frac{1}{n+1}, +\infty\right) = \bigwedge \{+\infty\} = +\infty.$$

Because multiplication is commutative the same value is assigned to the symmetric case  $(+\infty, 0)$ . □

**Lemma 18.22** (Upper pre-image of  $\text{mul}\downarrow$ ). For  $r \in \overline{\mathbb{R}}_{\geq 0}$ :

$$\mathbf{Ui} \text{ mul}\downarrow : r \mapsto \begin{cases} \uparrow\langle 0, 0 \rangle, & \text{if } r = 0, \\ \uparrow\{\langle +\infty, 0 \rangle, \langle 0, +\infty \rangle\} \cup \uparrow\{\langle x, y \rangle \in \mathbb{R}_{\geq 0}^2 : \text{mul}(x, y) = r\}, & \text{if } 0 < r < +\infty, \\ \uparrow\{\langle +\infty, 0 \rangle, \langle 0, +\infty \rangle\}, & \text{if } r = +\infty. \end{cases}$$

**Lemma 18.23** (Lower pre-image of  $\text{mul}\downarrow$ ). For  $s \in \overline{\mathbb{R}}_{\geq 0}$ :

$$\mathbf{Li} \text{ mul}\downarrow : s \mapsto \begin{cases} (\{0\} \times \downarrow\{+\infty\}) \cup (\downarrow\{+\infty\} \times \{0\}), & \text{if } s = 0, \\ (\{0\} \times \downarrow\{+\infty\}) \cup (\downarrow\{+\infty\} \times \{0\}) \cup \downarrow\{\langle x, y \rangle \in \mathbb{R}_{\geq 0}^2 : \text{mul}(x, y) = s\}, & \text{if } 0 < s < +\infty, \\ \downarrow\langle +\infty, +\infty \rangle, & \text{if } s = +\infty. \end{cases}$$

Note that in the case  $s = 0$  the downset cannot be written as the down closure of an antichain. We expect this because  $\text{mul}\downarrow$  is not Scott continuous.

### 18.4.3. Division

We consider division on the non-negative real numbers

$$\text{div} : \mathbb{R}_{\geq 0} \times \mathbb{R}_{> 0}^{\text{op}} \rightarrow_{\text{Pos}} \mathbb{R}_{\geq 0} \quad (60)$$

As for addition and multiplication, we wish to extend  $\text{div}$  to a map  $\overline{\mathbb{R}}_{\geq 0} \times \overline{\mathbb{R}}_{\geq 0}^{\text{op}} \rightarrow_{\text{Pos}} \overline{\mathbb{R}}_{\geq 0}$  that is either Scott continuous or Scott co-continuous.

**Lemma 18.24** (Scott continuous extension of division). There exists a Scott continuous map

$$\text{div}\uparrow : \overline{\mathbb{R}}_{\geq 0} \times \overline{\mathbb{R}}_{\geq 0}^{\text{op}} \rightarrow_{\text{Pos}} \overline{\mathbb{R}}_{\geq 0}$$

extending ordinary division on  $\mathbb{R}_{\geq 0} \times \mathbb{R}_{> 0}^{\text{op}}$ . It is uniquely determined by continuity and given case-wise by

$$\text{div}\uparrow(x, y) = \text{div}(x, y) \quad (x \in \mathbb{R}_{\geq 0}, y > 0), \quad (61)$$

$$\text{div}\uparrow(0, 0) = 0, \quad (62)$$

$$\text{div}\uparrow(x, 0) = +\infty \quad (x > 0), \quad (63)$$

$$\text{div}\uparrow(+\infty, y) = +\infty \quad (y > 0), \quad (64)$$

$$\text{div}\uparrow(x, +\infty) = 0 \quad (x < +\infty), \quad (65)$$

$$\text{div}\uparrow(+\infty, +\infty) = 0. \quad (66)$$

*This construction is described by the schema [M\\_DivideUConstant](#) (Section 26.3.38).*

*Proof.* First, we establish the values at the boundary points by continuity arguments:

**Case 1:**  $\text{div}\uparrow(0, 0)$ . Consider the directed set  $D = \{(0, y) : y > 0\}$ . In the product order  $\overline{\mathbb{R}}_{\geq 0} \times \overline{\mathbb{R}}_{\geq 0}^{\text{op}}$ , this set has supremum  $(0, 0)$  (since in the opposite order on the second coordinate, smaller  $y$  values are larger). For all  $(0, y) \in D$ , we have  $\text{div}\uparrow(0, y) = 0/y = 0$ . By Scott-continuity,

$$\text{div}\uparrow(0, 0) = \sup_{(0, y) \in D} \text{div}\uparrow(0, y) = \sup_{y > 0} 0 = 0.$$

**Case 2:**  $\text{div}\uparrow(x, 0)$  for  $x > 0$ . Consider the directed set  $D = \{(x, y) : 0 < y < 1\}$ . In  $\overline{\mathbb{R}}_{\geq 0} \times \overline{\mathbb{R}}_{\geq 0}^{\text{op}}$ , this set has supremum  $(x, 0)$  (since in the opposite order, 0 is the largest element). For any  $(x, y) \in D$ , we have  $\text{div}\uparrow(x, y) = x/y$ . As  $y \rightarrow 0^+$ , we have  $x/y \rightarrow +\infty$ . By

Scott-continuity in the second coordinate,

$$\text{div}\uparrow(x, 0) = \sup_{0 < y < 1} \text{div}\uparrow(x, y) = \sup_{0 < y < 1} \frac{x}{y} = +\infty.$$

**Case 3:  $\text{div}\uparrow(+\infty, y)$  for  $y > 0$ .** Consider the directed set  $D = \{(x, y) : x > 0\}$  with supremum  $(+\infty, y)$ . For any  $(x, y) \in D$ , we have  $\text{div}\uparrow(x, y) = x/y$ . By Scott-continuity in the first coordinate,

$$\text{div}\uparrow(+\infty, y) = \sup_{x > 0} \text{div}\uparrow(x, y) = \sup_{x > 0} \frac{x}{y} = +\infty.$$

**Case 4:  $\text{div}\uparrow(x, +\infty)$  for  $x < +\infty$ .** We need to find a directed set whose supremum is  $(x, +\infty)$ . Since we're working with  $\overline{\mathbb{R}}_{\geq 0} \times \overline{\mathbb{R}}_{\geq 0}^{\text{op}}$ , recall that in the opposite order,  $+\infty$  is the bottom element. Consider the increasing sequence  $y_n \rightarrow \infty$  in the usual order. Then  $(x, y_n)$  forms a directed set in the product topology, with supremum  $(x, +\infty)$ .

For each element, we have  $\text{div}\uparrow(x, y_n) = x/y_n$ . Taking the supremum as  $n \rightarrow \infty$ :

$$\text{div}\uparrow(x, +\infty) = \sup_n \text{div}\uparrow(x, y_n) = \sup_n \frac{x}{y_n} = 0.$$

**Case 5:  $\text{div}\uparrow(+\infty, +\infty)$ .** Take the directed chain  $D = \{(x, +\infty) : 0 < x < +\infty\}$ , ordered by the first coordinate (the second is fixed to the bottom element  $+\infty$ ). It is directed and  $\sup D = (+\infty, +\infty)$ . For every  $(x, +\infty) \in D$  we have  $\text{div}\uparrow(x, +\infty) = 0$  from Case 4, hence by Scott-continuity

$$\text{div}\uparrow(+\infty, +\infty) = \sup_{(x, +\infty) \in D} \text{div}\uparrow(x, +\infty) = 0.$$

□

**Lemma 18.25** (Scott co-continuous extension of division). There exists a Scott co-continuous map

$$\text{div}\downarrow : \overline{\mathbb{R}}_{\geq 0} \times \overline{\mathbb{R}}_{\geq 0}^{\text{op}} \rightarrow_{\text{Pos}} \overline{\mathbb{R}}_{\geq 0}$$

extending ordinary division on  $\mathbb{R}_{\geq 0} \times \mathbb{R}_{> 0}^{\text{op}}$ . It is uniquely determined by continuity and given case-wise by

$$\text{div}\downarrow(x, y) = \text{div}(x, y) \quad (x \in \mathbb{R}_{\geq 0}, y > 0), \quad (67)$$

$$\text{div}\downarrow(0, 0) = +\infty, \quad (68)$$

$$\text{div}\downarrow(x, 0) = +\infty \quad (x > 0), \quad (69)$$

$$\text{div}\downarrow(+\infty, y) = +\infty \quad (y > 0), \quad (70)$$

$$\text{div}\downarrow(x, +\infty) = 0 \quad (x < +\infty), \quad (71)$$

$$\text{div}\downarrow(+\infty, +\infty) = +\infty. \quad (72)$$

This construction is described by the schema [M\\_DivideLConstant](#) (Section 26.3.37).

*Proof.* We need to verify that the given extension  $\text{div}\downarrow$  is Scott co-continuous. Since we are working with the opposite order on the second coordinate (as indicated by  $\overline{\mathbb{R}}_{\geq 0} \times \overline{\mathbb{R}}_{\geq 0}^{\text{op}}$ ), we need to check that  $\text{div}\downarrow$  preserves infima of filtered sets.

First, we establish the values at the boundary points by co-continuity arguments:

**Case 1:  $\text{div}\downarrow(0, 0)$ .** Consider any filtered set  $F$  with  $\inf F = (0, 0)$  in  $\overline{\mathbb{R}}_{\geq 0} \times \overline{\mathbb{R}}_{\geq 0}^{\text{op}}$ . For the infimum of the first coordinates to be 0, we need elements  $(x_i, y_i) \in F$  with  $x_i \rightarrow 0$ . For the second coordinate in the opposite order,  $\inf_{\text{op}}\{y_i\} = 0$  means  $\sup\{y_i\} = 0$  in the usual order. Since  $y_i \geq 0$ , having  $\sup\{y_i\} = 0$  implies  $y_i = 0$  for all  $i$ . Thus, any such filtered set consists of elements of the form  $(x, 0)$  where  $x > 0$  and  $x \rightarrow 0$ . For all such elements,  $\text{div}\downarrow(x, 0) = +\infty$ . By Scott-co-continuity,

$$\text{div}\downarrow(0, 0) = \inf\{\text{div}\downarrow(x, 0) : (x, 0) \in F\} = \inf\{+\infty\} = +\infty.$$

**Case 2:  $\text{div}\downarrow(x, 0)$  for  $x > 0$ .** This is already defined as  $+\infty$  by the natural extension, since division by zero yields infinity.

**Case 3:  $\text{div}\downarrow(+\infty, y)$  for  $y > 0$ .** This follows directly from the fact that dividing infinity by any positive number yields infinity.

**Case 4:  $\text{div}\downarrow(x, +\infty)$  for  $x < +\infty$ .** Consider a filtered set  $F$  with  $\inf F = (x, +\infty)$  in  $\overline{\mathbb{R}}_{\geq 0} \times \overline{\mathbb{R}}_{\geq 0}^{\text{op}}$ . In the opposite order on the second coordinate,  $+\infty$  is the bottom element. For any filtered set converging to  $(x, +\infty)$ , we can consider  $F = \{(x, y) : y > M\}$  for large  $M$ . For

$(x, y) \in F$ , we have  $\text{div}\downarrow(x, y) = x/y \rightarrow 0$  as  $y \rightarrow \infty$ . By Scott-co-continuity,

$$\text{div}\downarrow(x, +\infty) = \inf_{y > M} \text{div}\downarrow(x, y) = \inf_{y > M} \frac{x}{y} = 0.$$

**Case 5:  $\text{div}\downarrow(+\infty, +\infty)$ .** Consider any filtered set  $F$  with  $\inf F = (+\infty, +\infty)$  in  $\overline{\mathbb{R}}_{\geq 0} \times \overline{\mathbb{R}}_{\geq 0}^{\text{op}}$ . For the infimum of the first coordinates to be  $+\infty$ , all elements must have first coordinate equal to  $+\infty$ . Thus  $F$  consists of elements of the form  $(+\infty, y)$  where  $y > 0$ . For all such elements,  $\text{div}\downarrow(+\infty, y) = +\infty$  from Case 3. By Scott-co-continuity,

$$\text{div}\downarrow(+\infty, +\infty) = \inf\{\text{div}\downarrow(+\infty, y) : (+\infty, y) \in F\} = \inf\{+\infty\} = +\infty.$$

The verification that  $\text{div}\downarrow$  preserves infima of all filtered sets follows from these boundary cases and the continuity of ordinary division on the interior of the domain.  $\square$

## 18.5. Unary join and meet operations

In this section we look at the unary meet and join, compute their Scott-continuity properties and their upper and lower inverses.

### Definition 18.26 (Unary meet)

Given a poset  $\mathbf{P}$ , the unary meet  $\wedge_c$ , is defined whenever  $c$  is such that  $x \wedge c$  exists for all  $x \in \mathbf{P}$ .

$$\begin{array}{ccc} \text{M\_MeetConstant}(\mathbf{P}, c) : \mathbf{P} & \xrightarrow{\text{Pos}} & \mathbf{P} \\ x & \longmapsto & x \wedge c \end{array} \quad (73)$$

This construction is described by the schema [M\\_MeetConstant](#) (Section 26.3.13).

### Definition 18.27 (Unary join)

The unary join  $\vee_c$  is defined whenever  $c$  is such that  $x \vee c$  exists for all  $x \in \mathbf{P}$ .

$$\begin{array}{ccc} \text{M\_JoinConstant}(\mathbf{P}, c) : \mathbf{P} & \xrightarrow{\text{Pos}} & \mathbf{P} \\ x & \longmapsto & x \vee c \end{array} \quad (74)$$

This construction is described by the schema [M\\_JoinConstant](#) (Section 26.3.11).

We can prove that the unary join is Scott continuous in a dcpo directly.

**Lemma 18.28** (Join with a constant in dcpo). If  $\mathbf{P}$  is a dcpo,  $c \in \mathbf{P}$  is such that  $x \vee c$  exists for all  $x \in \mathbf{P}$ , the map

$$\begin{array}{ccc} \vee_c : \mathbf{P} & \xrightarrow{\text{Pos}} & \mathbf{P} \\ x & \longmapsto & x \vee c \end{array}$$

is Scott continuous.

*Proof.* Let  $\mathbf{D} \subseteq \mathbf{P}$  be directed. Then we have that

$$\vee_c \left( \bigvee^{\uparrow} \mathbf{D} \right) \doteq \left( \bigvee^{\uparrow} \mathbf{D} \right) \vee c \stackrel{(a)}{=} \bigvee_{d \in \mathbf{D}}^{\uparrow} (d \vee c) = \bigvee^{\uparrow} \vee_c [\mathbf{D}].$$

where (a) follows from Lemma 18.29 and Lemma 18.31.  $\square$

**Lemma 18.29.** Let  $\mathbf{P} = \langle \mathbf{P}, \leq \rangle$  be a poset. Fix  $a \in \mathbf{P}$  and  $\mathbf{B} \subseteq \mathbf{P}$ . If these joins exist:

$$\bigvee_{b \in \mathbf{B}} b, \quad (75)$$

$$(a \vee b) \quad \forall b \in \mathbf{B} \quad (76)$$

$$\bigvee_{b \in \mathbf{B}} (a \vee b) \quad (77)$$

$$a \vee \bigvee_{b \in \mathbf{B}} b, \quad (78)$$

then the following equality holds:

$$a \vee \bigvee_{b \in \mathbf{B}} b = \bigvee_{b \in \mathbf{B}} (a \vee b) \quad (79)$$

*Proof.* Call

$$t \doteq \bigvee_{b \in \mathbf{B}} b \quad s \doteq a \vee \bigvee_{b \in \mathbf{B}} b = a \vee t \quad u \doteq \bigvee_{b \in \mathbf{B}} (a \vee b).$$

We need to prove that  $s = u$ . From  $b \leq t$ , we get  $a \vee b \leq a \vee t$ , and so  $u \leq s$ .

First note two elementary inequalities that follow from the definition of  $u$ :

$$\begin{aligned} a &\leq u \quad (\text{since } a \leq a \vee b \leq u \text{ for every } b \in \mathbf{B}), \\ t &= \bigvee_{b \in \mathbf{B}} b \leq \bigvee_{b \in \mathbf{B}} (a \vee b) = u \quad (\text{because } b \leq a \vee b \leq u \text{ for each } b \in \mathbf{B}). \end{aligned}$$

Hence both  $a$  and  $t$  are below  $u$ , making  $u$  an *upper bound* of the pair  $\{a, t\}$ . But  $s = a \vee t$  is, by definition, the *least* such upper bound, so necessarily  $s \leq u$ . □

**Lemma 18.30.** If all meets exist, then the following equality holds:

$$a \wedge \bigwedge_{b \in \mathbf{B}} b = \bigwedge_{b \in \mathbf{B}} (a \wedge b) \quad (80)$$

*Proof.* Dual of Lemma 18.29. □

**Lemma 18.31.** Assume  $f : \mathbf{P} \rightarrow \mathbf{Q}$  is monotone. If  $\mathbf{D} \subseteq \mathbf{P}$  is directed,  $f[\mathbf{D}]$  is directed. If  $\mathbf{F} \subseteq \mathbf{P}$  is filtered,  $f[\mathbf{F}]$  is filtered.

**Lemma 18.32.**

$$\begin{aligned} \text{Li} \downarrow_c : \mathbf{P} &\rightarrow_{\text{PosL}} \mathbf{P} \\ r &\longmapsto \begin{cases} \downarrow r & \text{if } c \leq_P y \\ \emptyset & \text{otherwise} \end{cases} \end{aligned} \quad (81)$$

*Proof.* We compute

$$\text{Li} \downarrow_c(y) = \{x \in \mathbf{P} : x \vee c \leq y\} \quad (82)$$

In the case  $c \leq y$ , we have that  $x \vee c \leq y$  iff  $x \leq y$ . If  $c \not\leq y$ , there is no  $x$  such that  $x \vee c \leq y$ , because otherwise  $c \leq x \vee c \leq y$ . Therefore,  $\text{Li} \downarrow_c(y) = \downarrow y$  if  $c \leq y$  and  $\emptyset$  otherwise. □

Once we have the lower inverse, we can also recover the Scott-continuity property from Theorem 12.30.

**Corollary 18.33.**  $\downarrow_c$  is Scott continuous.

*Proof.* From Theorem 12.31 and Lemma 18.32. Note that  $\emptyset = \downarrow \emptyset$ . □

**Lemma 18.34.**

$$\begin{aligned} \text{Ui} \downarrow_c : \mathbf{P} &\rightarrow_{\text{PosU}} \mathbf{P} \\ f^* &\longmapsto \begin{cases} \mathbf{P} & \text{if } f \leq_P c \\ \uparrow f^* & \text{otherwise} \end{cases} \end{aligned} \quad (83)$$

*Proof.* We compute

$$\text{Ui} \downarrow_c(f) = \{x \in \mathbf{P} : x \vee c \geq f\} \quad (84)$$

(a) In the case  $f \leq c$ , we have that  $x \vee c \geq f$  is always true.

(b) In the case  $f \not\leq c$ , we note that because  $f \leq x \vee c$ ,  $f$  must be below either  $c$  or  $x$ . But it is not below  $c$ , so it must be below  $x$ . So we have  $x \geq f$ . Conversely, if  $x \geq f$ , then  $x \vee c \geq f$ . □

**Corollary 18.35.** If  $\mathbf{P}$  is fBWF then  $\downarrow_c$  is Scott co-continuous.

*Proof.* From Theorem 12.35 and Lemma 18.34. □

**Lemma 18.36** (Meet with a constant in fcpo). If  $\mathbf{P}$  is a fcpo and  $c \in \mathbf{P}$  is such that  $x \wedge c$  exists for all  $x \in \mathbf{P}$ , the map

$$\begin{aligned} \wedge_c : \mathbf{P} &\rightarrow_{\text{Pos}} \mathbf{P} \\ x &\longmapsto x \wedge c \end{aligned}$$

is Scott co-continuous.

*Proof.* Dual of Lemma 18.28. □

**Lemma 18.37.**

$$\begin{aligned} \mathbf{Ui} \wedge_c : \mathbf{P} &\rightarrow_{\text{PosU}} \mathbf{P} \\ f^* &\longmapsto \begin{cases} \uparrow f^* & \text{if } f^* \leq_P c \\ \emptyset & \text{otherwise} \end{cases} \end{aligned} \quad (85)$$

*Proof.* Dual of Lemma 18.32. □

**Corollary 18.38.**  $\wedge_c$  is Scott co-continuous.

*Proof.* From Theorem 12.35 and Lemma 18.37. □

**Lemma 18.39.**

$$\begin{aligned} \mathbf{Li} \wedge_c : \mathbf{P} &\rightarrow_{\text{PosL}} \mathbf{P} \\ r &\longmapsto \begin{cases} \mathbf{P} & \text{if } c \leq_P r \\ \downarrow r & \text{otherwise} \end{cases} \end{aligned} \quad (86)$$

*Proof.* Dual of Lemma 18.34. □

**Corollary 18.40.** If  $\mathbf{P}$  is fAWF then  $\wedge_c$  is Scott continuous.

*Proof.* From Theorem 12.31 and Lemma 18.39. □

## 18.6. $n$ -ary joins and meets

In this section we look at the  $n$ -ary joins and meets.

### 18.6.1. $n$ -ary Join

#### Definition 18.41

Given  $n$  posets  $\llbracket \mathbf{P}_k \rrbracket$  all subposets of a join semilattice  $\mathbf{P}$ , we define the map

$$\begin{aligned} \mathbf{M\_Join}(\mathbf{P}, \llbracket \mathbf{P}_k \rrbracket) : \mathbf{P\_C\_Product}(\llbracket \mathbf{P}_k \rrbracket) &\rightarrow_{\text{Pos}} \mathbf{P} \\ \langle x_1, \dots, x_n \rangle &\longmapsto x_1 \vee x_2 \cdots \vee x_n \end{aligned} \quad (87)$$

*This construction is described by the schema  $\mathbf{M\_Join}$  (Section 26.3.10).*

We now analyze the case of binary joins, with  $n$ -ary joins an immediate generalization.

**Lemma 18.42** (Lower pre-image of binary join). The lower pre-image of the binary join operation  $\vee : \mathbf{P} \times \mathbf{P} \rightarrow_{\text{Pos}} \mathbf{P}$  is given by

$$\begin{aligned} \mathbf{Li} \vee : \mathbf{P} &\rightarrow_{\text{PosL}} \mathbf{P} \times \mathbf{P} \\ q &\longmapsto \downarrow \langle q, q \rangle \end{aligned} \quad (88)$$

Therefore,  $\mathbf{Li} \vee$  is fAWF with  $(\mathbf{Li} \vee)(q) = \downarrow \langle q, q \rangle$ .

*Proof.* We have  $(\mathbf{Li}\vee)(q) = \{\langle x, y \rangle \in \mathbf{P} \times \mathbf{P} \mid x \vee y \leq q\}$ . If  $x \vee y \leq q$ , then since  $x \leq x \vee y$  and  $y \leq x \vee y$ , we have  $x \leq q$  and  $y \leq q$ . Conversely, if  $x \leq q$  and  $y \leq q$ , then  $q$  is an upper bound of  $\{x, y\}$ , so  $x \vee y \leq q$ .  $\square$

**Lemma 18.43** (Upper pre-image of binary join). The upper pre-image of the binary join operation  $\vee : \mathbf{P} \times \mathbf{P} \rightarrow_{\text{Pos}} \mathbf{P}$  is given by

$$\begin{aligned} \mathbf{Ui}\vee : \mathbf{P} &\rightarrow_{\text{PosU}} \mathbf{P} \times \mathbf{P} \\ p &\longmapsto (\uparrow p \times \mathbf{P}) \cup (\mathbf{P} \times \uparrow p) \end{aligned} \quad (89)$$

*Proof.* We have  $(\mathbf{Ui}\vee)(p) = \{\langle x, y \rangle \in \mathbf{P} \times \mathbf{P} \mid p \leq x \vee y\}$ . If  $p \leq x \vee y$ , then we must have either  $p \leq x$  or  $p \leq y$  (or both). If  $p \leq x$ , then  $p \leq x \leq x \vee y$ . If  $p \leq y$ , then  $p \leq y \leq x \vee y$ . Conversely, if  $p \not\leq x$  and  $p \not\leq y$ , then  $p$  cannot be below their join.  $\square$

### 18.6.2. $n$ -ary Meet

#### Definition 18.44

Given  $n$  posets  $\llbracket \mathbf{P}_k \rrbracket$  all subposets of a meet semilattice  $\mathbf{P}$ , we define the map

$$\begin{aligned} \mathbf{M\_Meet}(\mathbf{P}, \llbracket \mathbf{P}_k \rrbracket) : \mathbf{P\_C\_Product}(\llbracket \mathbf{P}_k \rrbracket) &\rightarrow_{\text{Pos}} \mathbf{P} \\ \langle x_1, \dots, x_n \rangle &\longmapsto x_1 \wedge x_2 \wedge \dots \wedge x_n \end{aligned} \quad (90)$$

*This construction is described by the schema  $\mathbf{M\_Meet}$  (Section 26.3.12).*

**Lemma 18.45** (Upper pre-image of binary meet). The upper pre-image of the binary meet operation  $\wedge : \mathbf{P} \times \mathbf{P} \rightarrow_{\text{Pos}} \mathbf{P}$  is given by

$$\mathbf{Ui}\wedge : p \longmapsto \uparrow \langle p, p \rangle \quad (91)$$

Therefore,  $\mathbf{Ui}\wedge$  is fBWF.

*Proof.* We compute  $(\mathbf{Ui}\wedge)(p) = \{\langle x, y \rangle \in \mathbf{P} \times \mathbf{P} \mid p \leq x \wedge y\}$ . If  $p \leq x \wedge y$ , then since  $x \wedge y \leq x$  and  $x \wedge y \leq y$ , we have  $p \leq x$  and  $p \leq y$ . Conversely, if  $p \leq x$  and  $p \leq y$ , then  $p$  is a lower bound of  $\{x, y\}$ , so  $p \leq x \wedge y$ .  $\square$

**Lemma 18.46** (Lower pre-image of binary meet). The lower pre-image of the binary meet operation  $\wedge : \mathbf{P} \times \mathbf{P} \rightarrow_{\text{Pos}} \mathbf{P}$  is given by

$$\mathbf{Li}\wedge : q \longmapsto (\downarrow q \times \mathbf{P}) \cup (\mathbf{P} \times \downarrow q) \quad (92)$$

*Proof.* We compute  $(\mathbf{Li}\wedge)(q) = \{\langle x, y \rangle \in \mathbf{P} \times \mathbf{P} \mid x \wedge y \leq q\}$ . If  $x \wedge y \leq q$ , then we must have either  $x \leq q$  or  $y \leq q$  (or both). If  $x \leq q$ , then  $x \wedge y \leq x \leq q$ . If  $y \leq q$ , then  $x \wedge y \leq y \leq q$ . The converse clearly holds.  $\square$

### 18.7. Lifts to subsets

#### Definition 18.47

Given a map  $f : \mathbf{P} \rightarrow_{\text{Pos}} \mathbf{Q}$ , we define

$$\begin{aligned} \mathbf{M\_C\_LiftToSubsets}(f) : \mathbf{P\_C\_Power}(\mathbf{P}) &\rightarrow_{\text{Pos}} \mathbf{P\_C\_Power}(\mathbf{Q}) \\ \mathbf{A} &\longmapsto \bigcup_{x \in \mathbf{A}} f(x) \end{aligned} \quad (93)$$

*This construction is described by the schema  $\mathbf{M\_C\_LiftToSubsets}$  (Section 26.3.52).*

#### Definition 18.48

Given a map  $f : \mathbf{P} \rightarrow_{\text{Pos}} \mathbf{Q}$ , we define

$$\begin{aligned} \mathbf{M\_LiftToLowerSets}(f) : \mathbf{P\_C\_LowerSets}(\mathbf{P}) &\rightarrow_{\text{Pos}} \mathbf{P\_C\_LowerSets}(\mathbf{Q}) \\ \mathbf{A} &\longmapsto \bigcup_{x \in \mathbf{A}} f(x) \end{aligned} \quad (94)$$



This construction is described by the schema [M\\_LiftToLowerSets](#) (Section 26.3.53).

**Definition 18.49**

Given a map  $f : P \rightarrow_{\text{Pos}} Q$ , we define

$$\begin{array}{ccc} \text{M\_LiftToUpperSets}(f) : P\_C\_UpperSets(P) & \xrightarrow{\text{Pos}} & P\_C\_UpperSets(Q) \\ A & \longmapsto & \bigcup_{x \in A} f(x) \end{array} \quad (95)$$

This construction is described by the schema [M\\_LiftToUpperSets](#) (Section 26.3.54).

## 18.8. Plumbing

**Definition 18.50**

Given a poset  $P$  we define

$$\begin{array}{ccc} \text{M\_Lift} : P & \xrightarrow{\text{Pos}} & P\_C\_ProductSmash([P]) \\ p & \longmapsto & [p] \end{array} \quad (96)$$

This construction is described by the schema [M\\_Lift](#) (Section 26.3.60).

**Definition 18.51**

Given a poset  $P$  we define

$$\begin{array}{ccc} \text{M\_Unlift} : P\_C\_ProductSmash([P]) & \xrightarrow{\text{Pos}} & P \\ [p] & \longmapsto & p \end{array} \quad (97)$$

This construction is described by the schema [M\\_Unlift](#) (Section 26.3.63).

### 18.8.1. Slicing

**Definition 18.52**

Given a list of  $n$  posets  $[P_k]$  and an index  $j \in [1, \dots, n]$ , we define

$$\begin{array}{ccc} \text{M\_TakeIndex}([P_k], j) : P\_C\_Product([P_k]) & \xrightarrow{\text{Pos}} & P_j \\ \langle x_1, \dots, x_n \rangle & \longmapsto & x_j \end{array} \quad (98)$$

This construction is described by the schema [M\\_TakeIndex](#) (Section 26.3.61).

Analogously, [M\\_TakeRange](#) is a slicing operation from a smash product.

This construction is described by the schema [M\\_TakeRange](#) (Section 26.3.62).

### 18.8.2. Injections

**Definition 18.53**

Given a list of  $n$  posets  $[P_k]$  and an index  $j \in [1, \dots, n]$ , we define

$$\begin{array}{ccc} \text{M\_Injection}([P_k], j) : P_j & \xrightarrow{\text{Pos}} & P\_C\_Sum([P_k]) \\ x & \longmapsto & \langle j, x \rangle \end{array} \quad (99)$$

This construction is described by the schema [M\\_Injection](#) (Section 26.3.9).

Analogously, [M\\_SmashInjection](#) is the injection into  $P\_C\_Sum(\llbracket P_k \rrbracket)$ .

This construction is described by the schema [M\\_SmashInjection](#) (Section 26.3.16).

## 18.9. Catalog

### Definition 18.54

Given two posets  $P$  and  $Q$  and a set of input-output pairs  $\llbracket \langle p_k, q_k \rangle \rrbracket$ , we define

$$\begin{aligned} \text{M\_Explicit}(P, Q, \llbracket \langle p_k, q_k \rangle \rrbracket) : \cup \{p_k\} &\xrightarrow{\text{Pos}} Q \\ p_k &\longmapsto q_k \end{aligned} \quad (100)$$

This construction is described by the schema [M\\_Explicit](#) (Section 26.3.3).

## 18.10. Threshold maps

### Definition 18.55

Given two posets  $P$  and  $Q$  with top and bottom, we define

$$\begin{aligned} \text{M\_BottomIfNotTop}(P, Q) : P &\xrightarrow{\text{Pos}} Q \\ x &\longmapsto \begin{cases} T_Q & \text{if } T_P \leq x \\ \perp_Q & \text{if otherwise} \end{cases} \end{aligned} \quad (101)$$

This construction is described by the schema [M\\_BottomIfNotTop](#) (Section 26.3.55).

### Definition 18.56

Given two posets  $P$  and  $Q$  with top and bottom, we define

$$\begin{aligned} \text{M\_TopIfNotBottom}(P, Q) : P &\xrightarrow{\text{Pos}} Q \\ x &\longmapsto \begin{cases} \perp_Q & \text{if } x \leq \perp_P \\ T_Q & \text{otherwise} \end{cases} \end{aligned} \quad (102)$$

This construction is described by the schema [M\\_TopIfNotBottom](#) (Section 26.3.59).

### Definition 18.57

Given a poset  $P$  and two values  $T \leq_P V$ , we define

$$\begin{aligned} \text{M\_IdentityBelowThreshold}(P, T, V) : P &\xrightarrow{\text{Pos}} Q \\ x &\longmapsto \begin{cases} V & \text{if } T \leq x \\ x & \text{otherwise} \end{cases} \end{aligned} \quad (103)$$

This construction is described by the schema [M\\_IdentityBelowThreshold](#) (Section 26.3.56).

### Definition 18.58

Given a poset  $\mathbf{P}$  and a value  $V$ , we define

$$\begin{aligned} \mathbf{M\_Threshold1}(\mathbf{P}, V) : \mathbf{P} &\rightarrow_{\text{Pos}} \mathbf{P} \\ x &\mapsto \begin{cases} x & \text{if } x \leq V \\ v & \text{otherwise} \end{cases} \end{aligned} \quad (104)$$

This construction is described by the schema  $\mathbf{M\_Threshold1}$  (Section 26.3.57).

Note that if  $\mathbf{P}$  was a complete total order, this would be simplified to  $x \mapsto \min(x, V)$ . This map appears as the solution of the  $\mathbf{RF}$  query for  $\mathbf{DP\_FuncNotMoreThan}$ .

**Definition 18.59**

Given a poset  $\mathbf{P}$  and a value  $V$ , we define

$$\begin{aligned} \mathbf{M\_Threshold2}(\mathbf{P}, V) : \mathbf{P} &\rightarrow_{\text{Pos}} \mathbf{P} \\ x &\mapsto \begin{cases} x & \text{if } V \leq x \\ V & \text{otherwise} \end{cases} \end{aligned} \quad (105)$$

This construction is described by the schema  $\mathbf{M\_Threshold2}$  (Section 26.3.58).

Note that if  $\mathbf{P}$  was a complete total order, this would be simplified to  $x \mapsto \min(x, V)$ . This map appears as the solution of the  $\mathbf{FR}$  query for  $\mathbf{DP\_ResNotLessThan}$ .

## 18.11. Tests

### 18.11.1. constant $\leq x$

**Definition 18.60**

Given a poset  $\mathbf{P}$  and a constant  $c^* \in \mathbf{P}^{\text{op}}$ , we define

$$\begin{aligned} \mathbf{M\_C\_Leq\_X}(\mathbf{P}, c^*) : \mathbf{P} &\rightarrow_{\text{Pos}} \mathbf{Bool} \\ p &\mapsto c^* \leq_P p \end{aligned} \quad (106)$$

This construction is described by the schema  $\mathbf{M\_C\_Leq\_X}$  (Section 26.3.64).

**Lemma 18.61.**  $\mathbf{M\_C\_Leq\_X}(\mathbf{P}, c^*)$  is Scott co-continuous.

It is not Scott continuous in general.

*Proof.* The  $\mathbf{Ui}$  of this map is fBWF. □

We now give the upper and lower images of  $\mathbf{M\_C\_Leq\_X}$ .

First note that for all maps  $g : \mathbf{Q} \rightarrow_{\text{Pos}} \mathbf{Bool}$ , we have  $[\mathbf{Ui}g](\perp) = [\mathbf{Lig}](\top) = \mathbf{Q}$ . Moreover, we have  $[\mathbf{Ui}g](\top) = \mathbf{Q} \setminus [\mathbf{Lig}](\perp)$ . With this observation, we only need to compute  $[\mathbf{Ui}g](\top)$  or, equivalently,  $[\mathbf{Lig}](\perp)$ , and the rest follows.

**Lemma 18.62** (Upper / lower image of  $\mathbf{M\_C\_Leq\_X}$ ).

$$\begin{aligned} \mathbf{UiM\_C\_Leq\_X}(\mathbf{P}, c^*) : \mathbf{Bool} &\rightarrow_{\text{PosU}} \mathbf{P} \\ \perp &\mapsto \mathbf{P} \\ \top &\mapsto \uparrow_P c^* \end{aligned} \quad (107)$$

$$\begin{aligned} \mathbf{LiM\_C\_Leq\_X}(\mathbf{P}, c^*) : \mathbf{Bool} &\rightarrow_{\text{PosL}} \mathbf{P} \\ \perp &\mapsto \mathbf{P} \setminus (\uparrow_P c^*) \\ \top &\mapsto \mathbf{P} \end{aligned} \quad (108)$$

*Proof.* Let  $f(p) \doteq (c^* \leq_P p)$ . By definition,

$$[\mathbf{Ui}f](T) = \{p \in P \mid c^* \leq_P p\} = \uparrow_P c^* \quad (109)$$

□

### 18.11.2. $\text{constant} < x$

#### Definition 18.63

Given a poset  $P$  and a constant  $c^* \in P^{\text{op}}$ , we define

$$\begin{aligned} \mathbf{M\_C\_Lt\_X}(P, c^*) : P &\rightarrow_{\text{Pos}} \mathbf{Bool} \\ p &\longmapsto c^* <_P p \end{aligned} \quad (110)$$

This construction is described by the schema  $\mathbf{M\_C\_Lt\_X}$  (Section 26.3.65).

**Lemma 18.64.**  $\mathbf{M\_C\_Lt\_X}(P, c^*)$  is **not** Scott continuous or Scott co-continuous in general. It is Scott continuous if  $P$  is a dcpo and a total order.

**Lemma 18.65** (Upper / lower image of  $\mathbf{M\_C\_Lt\_X}$ ).

$$\begin{aligned} \mathbf{UiM\_C\_Lt\_X}(P, c^*) : \mathbf{Bool} &\rightarrow_{\text{PosU}} P \\ \perp &\longmapsto P \\ \top &\longmapsto \uparrow_P c^* \end{aligned} \quad (111)$$

$$\begin{aligned} \mathbf{LiM\_C\_Lt\_X}(P, c^*) : \mathbf{Bool} &\rightarrow_{\text{PosL}} P \\ \perp &\longmapsto P \setminus \uparrow_P c^* \\ \top &\longmapsto P \end{aligned} \quad (112)$$

*Proof.* Let  $f(p) \doteq (c^* <_P p)$ . By definition,

$$[\mathbf{Ui}f](T) = \{p \in P \mid c^* <_P p\} = \uparrow_P c^* \quad (113)$$

□

### 18.11.3. $x \leq \text{constant}$

#### Definition 18.66

Given a poset  $P$  and a constant  $c \in P$ , we define

$$\begin{aligned} \mathbf{M\_X\_Leq\_C}(P, c) : P^{\text{op}} &\rightarrow_{\text{Pos}} \mathbf{Bool} \\ p^* &\longmapsto p^* \leq_P c \end{aligned} \quad (114)$$

This construction is described by the schema  $\mathbf{M\_X\_Leq\_C}$  (Section 26.3.66).

**Lemma 18.67.**  $\mathbf{M\_X\_Leq\_C}(P, c)$  is Scott co-continuous. It is not Scott continuous in general.

**Lemma 18.68** (Upper / lower image of  $\mathbf{M\_X\_Leq\_C}$ ).

$$\begin{aligned} \mathbf{UiM\_X\_Leq\_C}(P, c) : \mathbf{Bool} &\rightarrow_{\text{PosU}} P^{\text{op}} \\ \perp &\longmapsto P \\ \top &\longmapsto \uparrow_{P^{\text{op}}} c \end{aligned} \quad (115)$$

$$\begin{aligned} \mathbf{LiM\_X\_Leq\_C}(P, c) : \mathbf{Bool} &\rightarrow_{\text{PosL}} P^{\text{op}} \\ \perp &\longmapsto P \setminus (\uparrow_{P^{\text{op}}} c) \\ \top &\longmapsto P \end{aligned} \quad (116)$$

*Proof.* Let  $f(p) \doteq (p \leq_P c)$ . We compute

$$[\mathbf{U}i f](T) = \{p \in \mathbf{P} \mid p \leq_P c\} = \downarrow_P c = \uparrow_{P^{\text{op}}} c \quad (117)$$

□

#### 18.11.4. $x < \text{constant}$

##### Definition 18.69

Given a poset  $\mathbf{P}$  and a constant  $c \in \mathbf{P}$ , we define

$$\begin{aligned} \mathbf{M\_X\_Lt\_C}(\mathbf{P}, c) : \mathbf{P}^{\text{op}} &\rightarrow_{\text{Pos}} \mathbf{Bool} \\ p^* &\longmapsto p^* <_P c \end{aligned} \quad (118)$$

This construction is described by the schema  $\mathbf{M\_X\_Lt\_C}$  (Section 26.3.67).

**Lemma 18.70.**  $\mathbf{M\_X\_Lt\_C}(\mathbf{P}, c)$  is **not** Scott continuous or Scott co-continuous in general. It is Scott continuous if  $\mathbf{P}$  is a fcpo and a total order.

**Lemma 18.71** (Upper / lower image of  $\mathbf{M\_X\_Lt\_C}$ ).

$$\begin{aligned} \mathbf{U}i \mathbf{M\_X\_Lt\_C}(\mathbf{P}, c) : \mathbf{Bool} &\rightarrow_{\text{PosU}} \mathbf{P}^{\text{op}} \\ \perp &\longmapsto \mathbf{P} \\ \top &\longmapsto \uparrow_{P^{\text{op}}} c \end{aligned} \quad (119)$$

$$\begin{aligned} \mathbf{L}i \mathbf{M\_X\_Lt\_C}(\mathbf{P}, c) : \mathbf{Bool} &\rightarrow_{\text{PosL}} \mathbf{P}^{\text{op}} \\ \perp &\longmapsto \mathbf{P} \setminus (\uparrow_{P^{\text{op}}} c) \\ \top &\longmapsto \mathbf{P} \end{aligned} \quad (120)$$

*Proof.* Let  $f(p) \doteq (p < c)$ . We compute

$$[\mathbf{U}i f](T) = \{p \in \mathbf{P} \mid p < c\} = \downarrow_P c = \uparrow_{P^{\text{op}}} c \quad (121)$$

□

## 18.12. Lower/upper set containment tests

### 18.12.1. Lower set containment tests

##### Definition 18.72

Given a poset  $\mathbf{P}$  and a lower set  $\mathbf{A} \in \mathbf{LP}$ , we define

$$\begin{aligned} \mathbf{M\_ContainedInLowerSet}(\mathbf{P}, \mathbf{A}) : \mathbf{P}^{\text{op}} &\rightarrow_{\text{Pos}} \mathbf{Bool} \\ p^* &\longmapsto p^* \in \mathbf{A} \end{aligned} \quad (122)$$

This construction is described by the schema  $\mathbf{M\_ContainedInLowerSet}$  (Section 26.3.7).

Note that the function  $\mathbf{M\_ContainedInLowerSet}$  is antitone.

**Lemma 18.73.**  $\mathbf{M\_ContainedInLowerSet}(\mathbf{P}, \mathbf{A})$  is Scott co-continuous iff  $\mathbf{A}$  is closed under directed suprema. A sufficient condition is that  $\mathbf{A}$  is fAWF.

**Lemma 18.74** (Upper / lower image of  $\mathbf{M\_ContainedInLowerSet}$ ).

$$\begin{aligned} \mathbf{U}i \mathbf{M\_ContainedInLowerSet}(\mathbf{P}, \mathbf{A}) : \mathbf{Bool} &\rightarrow_{\text{PosU}} \mathbf{P}^{\text{op}} \\ \perp &\longmapsto \mathbf{P} \\ \top &\longmapsto \mathbf{A} \end{aligned} \quad (123)$$

$$\begin{aligned}
\text{Li M\_ContainedInLowerSet}(P, A) : \text{Bool} &\rightarrow_{\text{PosL}} P^{\text{op}} \\
\perp &\longmapsto P \setminus A \\
\top &\longmapsto P
\end{aligned} \tag{124}$$

### 18.12.2. Upper set containment tests

#### Definition 18.75

Given a poset  $P$  and an upper set  $A \in \mathbf{UP}$ , we define

$$\begin{aligned}
\text{M\_ContainedInUpperSet}(P, A) : P &\rightarrow_{\text{Pos}} \text{Bool} \\
p &\longmapsto p \in A
\end{aligned} \tag{125}$$

This construction is described by the schema  $\text{M\_ContainedInUpperSet}$  (Section 26.3.8).

**Lemma 18.76.**  $\text{M\_ContainedInUpperSet}(P, A)$  is Scott co-continuous iff  $A$  is closed under filtered infima. A sufficient condition is that  $A$  is fBWF.

**Lemma 18.77** (Upper / lower image of  $\text{M\_ContainedInUpperSet}$ ).

$$\begin{aligned}
\text{Ui M\_ContainedInUpperSet}(P, A) : \text{Bool} &\rightarrow_{\text{PosU}} P \\
\perp &\longmapsto P \\
\top &\longmapsto A
\end{aligned} \tag{126}$$

$$\begin{aligned}
\text{Li M\_ContainedInUpperSet}(P, A) : \text{Bool} &\rightarrow_{\text{PosL}} P \\
\perp &\longmapsto P \setminus A \\
\top &\longmapsto P
\end{aligned} \tag{127}$$

### 18.13. Order as a function

#### Definition 18.78

Given a poset  $P$  we define

$$\begin{aligned}
\text{M\_Leq}(P) : P\_C\_Product([P^{\text{op}}, P]) &\rightarrow_{\text{Pos}} \text{Bool} \\
\langle x^*, y \rangle &\longmapsto x^* \leq_P y
\end{aligned} \tag{128}$$

This construction is described by the schema  $\text{M\_Leq}$  (Section 26.3.68).

**Lemma 18.79.**  $\text{M\_Leq}(P)$  is neither Scott co-continuous nor Scott continuous in general, not even on total orders.

## 19. Monotone map compositions catalog

### 19.1. Constructions for single maps

#### 19.1.1. Opposite of a map

**Definition 19.1**

Given a map  $f : P \rightarrow_{\text{Pos}} Q$ , the *opposite* of  $f$  is the map  $\text{M\_C\_Op}(f) : P^{\text{op}} \rightarrow_{\text{Pos}} Q^{\text{op}}$  such that:

$$\text{M\_C\_Op}(f) : x \mapsto f(x) \quad (1)$$

*This construction is described by the schema [M\\_C\\_Op](#) (Section 26.3.17).*

**Lemma 19.2.** If  $f$  is Scott continuous, then  $\text{M\_C\_Op}(f)$  is Scott co-continuous, and viceversa.

### 19.2. Constructions for multiple maps

#### 19.2.1. Parallel composition

**Definition 19.3** (Parallel composition)

Given a list of  $n$  maps  $\llbracket f_k : P_k \rightarrow_{\text{Pos}} Q_k \rrbracket$  the *parallel composition* of  $f_k$  is the map

$$\begin{aligned} \text{M\_C\_Parallel}(\llbracket f_k \rrbracket) : P\_C\_Product(\llbracket P_k \rrbracket) &\rightarrow_{\text{Pos}} P\_C\_Product(\llbracket Q_k \rrbracket) \\ \langle x_1, \dots, x_n \rangle &\longmapsto \langle f_1(x_1), \dots, f_n(x_n) \rangle \end{aligned} \quad (2)$$

*This construction is described by the schema [M\\_C\\_Parallel](#) (Section 26.3.25).*

**Lemma 19.4.** [M\\_C\\_Parallel](#) inherits the Scott flavor of the constituent maps: if all  $f_k$  are Scott continuous (Scott co-continuous), then [M\\_C\\_Parallel](#) is Scott continuous (Scott co-continuous).

**Lemma 19.5.**

$$\mathbf{Ui} \text{M\_C\_Parallel}(\llbracket f_k \rrbracket) = \mathbf{U1\_C\_Parallel}(\llbracket \mathbf{Ui} f_k \rrbracket) \quad (3)$$

$$\mathbf{Li} \text{M\_C\_Parallel}(\llbracket f_k \rrbracket) = \mathbf{L1\_C\_Parallel}(\llbracket \mathbf{Li} f_k \rrbracket) \quad (4)$$

There is a variant of the parallel composition that is associative.

**Definition 19.6** (Smash parallel composition)

Given a list of  $n$  maps  $\llbracket f_k : P_k \rightarrow_{\text{Pos}} Q_k \rrbracket$  the *smash parallel composition* is

$$\begin{aligned} \text{M\_C\_ParallelSmash}(\llbracket f_k \rrbracket) : P\_C\_ProductSmash(\llbracket P_k \rrbracket) &\rightarrow_{\text{Pos}} P\_C\_ProductSmash(\llbracket Q_k \rrbracket) \\ [x_1 \mid \dots \mid x_n] &\longmapsto [f_1(x_1) \mid \dots \mid f_n(x_n)] \end{aligned} \quad (5)$$

*This construction is described by the schema [M\\_C\\_ParallelSmash](#) (Section 26.3.26).*

Finally, we can define hybrid versions of the product and smash product of maps that go from the product of domains to the smash product of codomains

$$\text{M\_C\_DomProdCodSmash}(\llbracket f_k \rrbracket) : P\_C\_Product(\llbracket P_k \rrbracket) \rightarrow_{\text{Pos}} P\_C\_ProductSmash(\llbracket Q_k \rrbracket) \quad (6)$$

This construction is described by the schema [M\\_C\\_DomProdCodSmash](#) (Section 26.3.22).

Or, viceversa:

$$\mathbf{M\_C\_DomSmashCodProd}(\llbracket f_k \rrbracket) : \mathbf{P\_C\_ProductSmash}(\llbracket \mathbf{P}_k \rrbracket) \rightarrow_{\mathbf{Pos}} \mathbf{P\_C\_Product}(\llbracket \mathbf{Q}_k \rrbracket) \quad (7)$$

This construction is described by the schema [M\\_C\\_DomSmashCodProd](#) (Section 26.3.23).

These are based on the isomorphism

$$\mathbf{P\_C\_ProductSmash}(\llbracket \mathbf{P}_k \rrbracket) \leftrightarrow_{\mathbf{Pos}} \mathbf{P\_C\_Product}(\llbracket \mathbf{P}_k \rrbracket) \quad (8)$$

### 19.2.2. Series composition

#### Definition 19.7

The series composition of a list of  $n$  maps  $\llbracket f_k : \mathbf{P}_k \rightarrow_{\mathbf{Pos}} \mathbf{Q}_k \rrbracket$  is defined whenever for  $k = 1, \dots, n-1$ ,  $\mathbf{Q}_k \subseteq \mathbf{P}_{k+1}$ . It is given by:

$$\begin{aligned} \mathbf{M\_C\_Series}(\llbracket f_k \rrbracket) : \mathbf{P}_1 &\rightarrow_{\mathbf{Pos}} \mathbf{Q}_n \\ x &\longmapsto f_n(f_{n-1}(\dots(f_1(x))\dots)) \end{aligned} \quad (9)$$

#### Lemma 19.8.

If all  $f_i$  are Scott continuous, then  $\mathbf{M\_C\_Series}(\llbracket f_k \rrbracket)$  is Scott continuous.

If all  $f_i$  are Scott co-continuous, then  $\mathbf{M\_C\_Series}(\llbracket f_k \rrbracket)$  is Scott co-continuous.

#### Lemma 19.9.

$$\mathbf{UiM\_C\_Series}(\llbracket f_k \rrbracket) = \mathbf{U1\_C\_Series}(\text{reversed}(\llbracket \mathbf{Ui} f_k \rrbracket)) \quad (10)$$

$$\mathbf{LiM\_C\_Series}(\llbracket f_k \rrbracket) = \mathbf{L1\_C\_Series}(\text{reversed}(\llbracket \mathbf{Li} f_k \rrbracket)) \quad (11)$$

$$(12)$$

This construction is described by the schema [M\\_C\\_Series](#) (Section 26.3.29).

### 19.2.3. Product of maps

#### Definition 19.10

Given a list of  $n$  maps  $\llbracket f_k : \mathbf{P} \rightarrow_{\mathbf{Pos}} \mathbf{Q}_k \rrbracket$  their *product* is the map

$$\begin{aligned} \mathbf{M\_C\_Product}(\llbracket f_k \rrbracket) : \mathbf{P} &\rightarrow_{\mathbf{Pos}} \mathbf{P\_C\_Product}(\llbracket \mathbf{Q}_k \rrbracket) \\ x &\longmapsto \langle f_1(x), \dots, f_n(x) \rangle \end{aligned} \quad (13)$$

This construction is described by the schema [M\\_C\\_Product](#) (Section 26.3.27).

**Lemma 19.11.** [M\\_C\\_Product](#) inherits the Scott flavor of the constituent maps:

#### Lemma 19.12.

$$\mathbf{UiM\_C\_Product}(\llbracket f_k \rrbracket) = \mathbf{U1\_C\_ProdIntersection}(\llbracket \mathbf{Ui} f_k \rrbracket) \quad (14)$$

$$\mathbf{LiM\_C\_Product}(\llbracket f_k \rrbracket) = \mathbf{L1\_C\_ProdIntersection}(\llbracket \mathbf{Li} f_k \rrbracket) \quad (15)$$

$$(16)$$

#### Definition 19.13

Given a list of  $n$  maps  $\llbracket f_k : \mathbf{P} \rightarrow_{\mathbf{Pos}} \mathbf{Q}_k \rrbracket$  their *smash product* is the map

$$\begin{aligned} \mathbf{M\_C\_ProductSmash}(\llbracket f_k \rrbracket) : \mathbf{P} &\rightarrow_{\mathbf{Pos}} \mathbf{P\_C\_ProductSmash}(\llbracket \mathbf{Q}_k \rrbracket) \\ x &\longmapsto [f_1(x) \mid \dots \mid f_n(x)] \end{aligned} \quad (17)$$

This construction is described by the schema [M\\_C\\_ProductSmash](#) (Section 26.3.28).



### 19.2.4. Sum of maps

#### Definition 19.14

Given a list of  $n$  maps  $\llbracket f_k : P_k \rightarrow_{\text{Pos}} Q_k \rrbracket$  the *sum* of  $f_k$  is the map

$$\begin{aligned} \text{M\_C\_Sum}(\llbracket f_k \rrbracket) : P\_C\_Sum(\llbracket P_k \rrbracket) &\rightarrow P\_C\_Sum(\llbracket Q_k \rrbracket), \\ \langle i, x_i \rangle &\mapsto \langle i, f_i(x_i) \rangle. \end{aligned} \quad (18)$$

This construction is described by the schema [M\\_C\\_Sum](#) (Section 26.3.30).

By isomorphism we also construct the map

$$\text{M\_C\_SumSmash}(\llbracket f_k \rrbracket) : P\_C\_SumSmash(\llbracket P_k \rrbracket) \rightarrow_{\text{Pos}} P\_C\_SumSmash(\llbracket Q_k \rrbracket) \quad (19)$$

This construction is described by the schema [M\\_C\\_SumSmash](#) (Section 26.3.31).

**Lemma 19.15.** [M\\_C\\_Sum](#) inherits the Scott flavor of the constituent maps:

**Lemma 19.16.**

$$\text{Ui M\_C\_Sum}(\llbracket f_k \rrbracket) = \text{U1\_C\_Sum}(\llbracket \text{Ui } f_k \rrbracket) \quad (20)$$

$$\text{Li M\_C\_Sum}(\llbracket f_k \rrbracket) = \text{L1\_C\_Sum}(\llbracket \text{Li } f_k \rrbracket) \quad (21)$$

*Proof.* Let  $g \doteq \text{M\_C\_Sum}(\llbracket f_k \rrbracket)$  and evaluate it at a point  $\langle j, y \rangle$  for  $y \in Q_j$ . We see that the points in the domain must be as well in the  $j$ -th component:

$$[\text{Ui } g](\langle j, y \rangle) \doteq \{x \text{ such that } \langle j, y \rangle \leq g(x)\} \quad (22)$$

$$= \{\langle j, \psi \rangle \text{ such that } \langle j, y \rangle \leq g(\langle j, \psi \rangle)\} \quad (23)$$

$$= \{\langle j, \psi \rangle \text{ such that } y \leq g_j(\psi)\} \quad (24)$$

$$= \{\langle j, \psi \rangle \text{ for } \psi \in \text{Ui } g_j(y)\} \quad (25)$$

$$(26)$$

□

### 19.2.5. Coproduct of maps

#### Definition 19.17

Given a list of  $n$  maps  $\llbracket f_k : P_k \rightarrow_{\text{Pos}} Q \rrbracket$  the *coproduct* of  $f_k$  is the map

$$\begin{aligned} \text{M\_C\_Coproduct}(\llbracket f_k \rrbracket) : P\_C\_Sum(\llbracket P_k \rrbracket) &\rightarrow_{\text{Pos}} Q \\ \langle i, x_i \rangle &\longmapsto f_i(x_i) \end{aligned} \quad (27)$$

This construction is described by the schema [M\\_C\\_Coproduct](#) (Section 26.3.20).

**Lemma 19.18.** [M\\_C\\_Coproduct](#) inherits the Scott flavor of the constituent maps.

**Lemma 19.19.**

$$\text{Ui M\_C\_Coproduct}(\llbracket f_k \rrbracket) = \text{U1\_CoproductCod}(\llbracket \text{Ui } f_k \rrbracket) \quad (28)$$

$$\text{Li M\_C\_Coproduct}(\llbracket f_k \rrbracket) = \text{L1\_CoproductCod}(\llbracket \text{Li } f_k \rrbracket) \quad (29)$$

$$(30)$$

By isomorphism we define the smash coproduct of maps:

$$\text{M\_C\_CoproductSmash}(\llbracket f_k \rrbracket) : P\_C\_SumSmash(\llbracket P_k \rrbracket) \rightarrow_{\text{Pos}} Q \quad (31)$$

This construction is described by the schema [M\\_C\\_CoproductSmash](#) (Section 26.3.21).

### 19.2.6. Domain union

**Definition 19.20** (Domain union of maps)

Given a list of  $n$  maps  $\llbracket f_k : P_k \rightarrow_{\text{Pos}} Q \rrbracket$  such that all  $P_k$  are subposets of  $\bar{P}$ , and whenever  $x \in P_i \cap P_j$  we have that  $f_i(x) = f_j(x)$ , then the domain union of the maps is defined as

$$\begin{array}{ccc} \text{M\_C\_DomUnion}(\llbracket f_k \rrbracket) : P\_C\_Union(\llbracket P_k \rrbracket) & \xrightarrow{\text{Pos}} & Q \\ x & \longmapsto & f_k(x), \text{ where } k \text{ is the smallest } k \text{ such that } x \in P_k \end{array} \quad (32)$$

*This construction is described by the schema [M\\_C\\_DomUnion](#) (Section 26.3.24).*

This is a definition that does not make much sense mathematically – if the maps are equal on the intersection, then they are the same map. However, when the maps are considered as “algorithms”, the domain union is the formalway to combine the algorithms for special cases into a single algorithm.

**Lemma 19.21.**

$$\text{Ui M\_C\_DomUnion}(\llbracket f_k \rrbracket) = \text{U1\_C\_Union}(\llbracket \text{Ui } f_k \rrbracket) \quad (33)$$

$$\text{Li M\_C\_DomUnion}(\llbracket f_k \rrbracket) = \text{L1\_C\_Union}(\llbracket \text{Li } f_k \rrbracket) \quad (34)$$

$$(35)$$

## 20. Pos $\mathbf{L}$ and Pos $\mathbf{U}$ catalog

### 20.1. Identity morphisms

The identity morphisms are equal to  $\downarrow \text{id}_P$  and  $\uparrow \text{id}_P$  maps on the posets.

**Definition 20.1**

Given a poset  $P$ , we define

$$\begin{aligned} \text{L1\_Identity}(P) : P &\rightarrow_{\text{PosL}} P \\ r &\longmapsto \downarrow r \end{aligned} \quad (1)$$

This construction is described by the schema  $\text{L1\_Identity}$  (Section 26.4.4).

**Definition 20.2**

Given a poset  $P$ , we define

$$\begin{aligned} \text{U1\_Identity}(P) : P &\rightarrow_{\text{PosU}} P \\ f^* &\longmapsto \uparrow f^* \end{aligned} \quad (2)$$

This construction is described by the schema  $\text{U1\_Identity}$  (Section 26.5.4).

### 20.2. Lifting maps

**Definition 20.3** (Lifting map to Pos $\mathbf{L}$ )

Given a map  $g : \text{kdom} \rightarrow_{\text{Pos}} \text{kcod}$  we can lift it to

$$\begin{aligned} \text{L1\_Lift}(g) : \text{kdom} &\rightarrow_{\text{PosL}} \text{kcod} \\ r &\longmapsto \downarrow g(r) \end{aligned} \quad (3)$$

This construction is described by the schema  $\text{L1\_Lift}$  (Section 26.4.28).

**Definition 20.4** (Lifting map to Pos $\mathbf{U}$ )

Given a map  $g : \text{kdom}^{\text{op}} \rightarrow_{\text{Pos}} \text{kcod}$  we can lift it to

$$\begin{aligned} \text{U1\_Lift}(g) : \text{kdom} &\rightarrow_{\text{PosU}} \text{kcod} \\ f^* &\longmapsto \uparrow g(f^*) \end{aligned} \quad (4)$$

This construction is described by the schema  $\text{U1\_Lift}$  (Section 26.5.28).

### 20.3. Catalog maps

**Definition 20.5** (L1 Catalog map)

Given two posets  $\mathbf{F}$  and  $\mathbf{R}$ , a catalog of options  $\llbracket \langle f_k, r_k^* \rangle \rrbracket$ , we define the map

$$\begin{aligned} \text{L1\_Catalog}(\llbracket \langle f_k, r_k^* \rangle \rrbracket) : \mathbf{R} &\rightarrow_{\text{PosL}} \mathbf{F} \\ r &\mapsto \bigcap_k \begin{cases} \{f_k\} & \text{if } r_k^* \leq r, \\ \emptyset & \text{otherwise} \end{cases} \end{aligned} \quad (5)$$

This construction is described by the schema  $\text{L1\_Catalog}$  (Section 26.4.6).

**Definition 20.6** (U1 Catalog map)

Given two posets  $\mathbf{F}$  and  $\mathbf{R}$ , a catalog of options  $\llbracket \langle f_k, r_k^* \rangle \rrbracket$ , we define the map

$$\begin{aligned} \text{U1\_Catalog}(\llbracket \langle f_k, r_k^* \rangle \rrbracket) : \mathbf{F} &\rightarrow_{\text{PosU}} \mathbf{R} \\ f^* &\mapsto \bigcup_k \begin{cases} \{r_k^*\} & \text{if } f^* \leq f_k, \\ \emptyset & \text{otherwise} \end{cases} \end{aligned} \quad (6)$$

This construction is described by the schema  $\text{U1\_Catalog}$  (Section 26.5.6).

## 20.4. Union and intersection of principal lower sets

**Definition 20.7** (Intersection of principal lower sets)

Given  $n$  posets  $\llbracket \mathbf{P}_k \rrbracket$  that are subposets of  $\mathbf{P}$ , we define the map

$$\begin{aligned} \text{L1\_IntersectionOfPrinLowerSets}(\llbracket \mathbf{P}_k \rrbracket) : \text{P\_C\_Product}(\llbracket \mathbf{P}_k \rrbracket) &\rightarrow_{\text{PosL}} \mathbf{P} \\ \langle r_1, \dots, r_n \rangle &\mapsto \bigcap_k \downarrow r_k \end{aligned} \quad (7)$$

This construction is described by the schema  $\text{L1\_IntersectionOfPrinLowerSets}$  (Section 26.4.7).

**Definition 20.8** (Intersection of principal upper sets)

Given  $n$  posets  $\llbracket \mathbf{P}_k \rrbracket$  that are subposets of  $\mathbf{P}$ , we define the map

$$\begin{aligned} \text{U1\_IntersectionOfPrinUpperSets}(\llbracket \mathbf{P}_k \rrbracket) : \text{P\_C\_Product}(\llbracket \mathbf{P}_k \rrbracket) &\rightarrow_{\text{PosU}} \mathbf{P} \\ \langle f_1^*, \dots, f_n^* \rangle &\mapsto \bigcup_k \uparrow f_k^* \end{aligned} \quad (8)$$

This construction is described by the schema  $\text{U1\_IntersectionOfPrinUpperSets}$  (Section 26.5.7).

**Definition 20.9** (Union of principal lower sets)

Given  $n$  posets  $\llbracket \mathbf{P}_k \rrbracket$  that are subposets of  $\mathbf{P}$ , we define the map

$$\begin{aligned} \text{L1\_UnionOfPrinLowerSets}(\llbracket \mathbf{P}_k \rrbracket) : \text{P\_C\_Product}(\llbracket \mathbf{P}_k \rrbracket) &\rightarrow_{\text{PosL}} \mathbf{P} \\ \langle r_1, \dots, r_n \rangle &\mapsto \bigcup_k \downarrow r_k \end{aligned} \quad (9)$$

This construction is described by the schema  $\text{L1\_UnionOfPrinLowerSets}$  (Section 26.4.9).

**Definition 20.10** (Union of principal upper sets)

Given  $n$  posets  $\llbracket \mathbf{P}_k \rrbracket$  that are subposets of  $\mathbf{P}$ , we define the map

$$\begin{aligned} \text{U1\_UnionOfPrinUpperSets}(\llbracket \mathbf{P}_k \rrbracket) : \mathbf{P\_C\_Product}(\llbracket \mathbf{P}_k \rrbracket) &\rightarrow_{\text{PosU}} \mathbf{P} \\ \langle f_1^*, \dots, f_n^* \rangle &\longmapsto \bigcup_k \uparrow f_k^* \end{aligned} \quad (10)$$

This construction is described by the schema `U1_UnionOfPrinUpperSets` (Section 26.5.9).

## 20.5. Representing principal lower and upper sets

### Definition 20.11

Given two posets `kdom` and `kcod` that are both subposets of a common ambient poset  $\mathbf{P}$ , we can define the map

$$\begin{aligned} \text{L1\_RepresentPrincipalLowerSet}(\mathbf{P}, \text{kdom}, \text{kcod}) : \text{kdom} &\rightarrow_{\text{PosL}} \text{kcod} \\ r &\longmapsto (\downarrow_P r) \cap \text{kcod} \end{aligned} \quad (11)$$

This construction is described by the schema `L1_RepresentPrincipalLowerSet` (Section 26.4.8).

The idea is that we want to “represent” the principal lower set  $\downarrow_P r$  in terms of the codomain `kcod`.

For example, consider the case

- `kdom` =  $2\mathbb{Z}$ , the even integers
- `kcod` =  $3\mathbb{Z}$ , the multiples of 3

They are both subposets of the ambient poset  $\mathbb{Z}$ . The map would associate each even integer  $x$  with the down closure of the largest multiple of 3 less than or equal to  $x$ :

$$\begin{aligned} 10 &\mapsto \downarrow 9 \\ 8 &\mapsto \downarrow 6 \\ 6 &\mapsto \downarrow 6 \\ 4 &\mapsto \downarrow 3 \\ 2 &\mapsto \downarrow 0 \\ 0 &\mapsto \downarrow 0 \end{aligned}$$

In general, the posets are not total orders, so the result is not necessarily a principal lower set.

### Definition 20.12

Given two posets `kdom` and `kcod` that are both subposets of a common ambient poset  $\mathbf{P}$ , we can define the map

$$\begin{aligned} \text{U1\_RepresentPrincipalUpperSet}(\mathbf{P}, \text{kdom}, \text{kcod}) : \text{kdom} &\rightarrow_{\text{PosU}} \text{kcod} \\ f^* &\longmapsto (\uparrow_P f^*) \cap \text{kcod} \end{aligned} \quad (12)$$

This construction is described by the schema `U1_RepresentPrincipalUpperSet` (Section 26.5.8).

## 20.6. Generic inverses for mathematical operations

In this section we discuss the schemas used to indicate the approximations of the upper/lower inverse of addition and multiplication.

Let  $\mathbf{P}$  be a poset in which there is defined an addition operation  $\text{add}^n : \mathbf{P}^n \times \mathbf{P} \rightarrow \mathbf{P}$ . In Section 18.4 we have discussed at length the cases of  $\mathbf{P} = \overline{\mathbb{R}}$ .

Consider the upper inverse of addition:

$$\begin{aligned} \text{Ui\_add}^n : \mathbf{P} &\rightarrow_{\text{PosU}} \mathbf{P}^n \\ f^* &\longmapsto \{ \langle x_1, \dots, x_n \rangle \text{ such that } f^* \leq \text{add}^n(x_1, \dots, x_n) \} \end{aligned} \quad (13)$$

The schemas `U1_InvSum_Pes` and `U1_InvSum_Opt` are not fully specified as function, but rather they are defined by the following properties.

**Definition 20.13**

Given a poset  $P$ , a dimensionality  $n \geq 2$  and a resolution  $r \in \mathbb{N}$ , we define  $\text{U1\_InvSum\_Pes}(P, n, r)$  as indicating any family of maps that satisfy the following properties any map that satisfies:

- It is a pessimistic approximation of  $\text{Ui add}^n$  for any fixed  $r$ :

$$\text{U1\_InvSum\_Pes}(P, n, r) \leq \text{Ui add}^n \quad (14)$$

- It is increasing in  $n$ :

$$\text{U1\_InvSum\_Pes}(P, n, r) \leq \text{U1\_InvSum\_Pes}(P, n, r + 1) \quad (15)$$

- It approximates  $\text{Ui add}^n$  in the limit:

$$\sup_r \text{U1\_InvSum\_Pes}(P, n, r) = \text{Ui add}^n \quad (16)$$

This construction is described by the schema  $\text{U1\_InvSum\_Pes}$  (Section 26.5.25).

The definition for  $\text{U1\_InvSum\_Opt}$  is analogous.

**Definition 20.14**

Given a poset  $P$ , a dimensionality  $n \geq 2$ , and a resolution  $r \in \mathbb{N}$ , we define  $\text{U1\_InvSum\_Opt}(P, n, r)$  as indicating any family of maps that satisfy the following properties any map that satisfies:

- It is an optimistic approximation of  $\text{Ui add}^n$  for any fixed  $n$ :

$$\text{Ui add}^r \leq \text{U1\_InvSum\_Opt}(P, n, r) \quad (17)$$

- It is decreasing in  $r$ :

$$\text{U1\_InvSum\_Opt}(P, n, r + 1) \leq \text{U1\_InvSum\_Opt}(P, n, r) \quad (18)$$

- It approximates  $\text{Ui add}^n$  in the limit:

$$\inf_r \text{U1\_InvSum\_Opt}(P, n, r) = \text{Ui add}^n \quad (19)$$

This construction is described by the schema  $\text{U1\_InvSum\_Opt}$  (Section 26.5.24).

The definition for  $\text{L1\_InvSum\_Pes}$  and  $\text{L1\_InvSum\_Opt}$  is analogous, replacing  $\text{Ui}$  with  $\text{Li}$ .

As for the maps  $\text{L1\_InvMul\_Opt}$ ,  $\text{U1\_InvMul\_Opt}$ ,  $\text{L1\_InvMul\_Pes}$ ,  $\text{U1\_InvMul\_Pes}$  the definition is analogous, replacing  $\text{add}$  with  $\text{mul}$ .

## 20.7. Filtering

**Definition 20.15**

Given a poset  $P$  and a monotone map  $g : P \rightarrow_{\text{Pos}} \mathbf{Bool}$ , we define

$$\begin{aligned} \text{L1\_FromFilter}(g) : P &\rightarrow_{\text{PosL}} P \\ r &\longmapsto \begin{cases} \{r\} & \text{if } g(r) \\ \emptyset & \text{otherwise} \end{cases} \end{aligned} \quad (20)$$

This construction is described by the schema  $\text{L1\_FromFilter}$  (Section 26.4.26).

**Definition 20.16**

Given a poset  $P$  and a monotone map  $g : P^{\text{op}} \rightarrow_{\text{Pos}} \mathbf{Bool}$ , we define

$$\begin{aligned} \text{U1\_FromFilter}(g) : P &\rightarrow_{\text{PosU}} P \\ f^* &\longmapsto \begin{cases} \{f^*\} & \text{if } g(f^*) \\ \emptyset & \text{otherwise} \end{cases} \end{aligned} \quad (21)$$

This construction is described by the schema **U1\_FromFilter** (Section 26.5.26).

## 20.8. Parallel composition

### Definition 20.17 (Parallel composition for **PosL**)

For a family of maps  $\llbracket \ell_k : \text{kdom}_k \rightarrow_{\text{PosL}} \text{kcod}_k \rrbracket$ , the monoidal product is defined as:

$$\begin{aligned} \text{L1\_C\_Parallel}(\llbracket \ell_k \rrbracket) : \text{P\_C\_Product}(\llbracket \text{kdom}_k \rrbracket) &\rightarrow_{\text{PosL}} \text{P\_C\_Product}(\llbracket \text{kcod}_k \rrbracket) \\ \langle r_1, \dots, r_n \rangle &\longmapsto \otimes_k \ell_k(r_k) \end{aligned} \quad (22)$$

where  $\otimes$  is set product.

This construction is described by the schema **L1\_C\_Parallel** (Section 26.4.13).

### Definition 20.18 (Parallel composition for **PosU**)

For a family of maps  $\llbracket u_k : \text{kdom}_k \rightarrow_{\text{PosU}} \text{kcod}_k \rrbracket$ , the monoidal product is defined as:

$$\begin{aligned} \text{U1\_C\_Parallel}(\llbracket u_k \rrbracket) : \text{P\_C\_Product}(\llbracket \text{kdom}_k \rrbracket) &\rightarrow_{\text{PosU}} \text{P\_C\_Product}(\llbracket \text{kcod}_k \rrbracket) \\ \langle f_1^*, \dots, f_n^* \rangle &\longmapsto \otimes_k u_k(f_k^*) \end{aligned} \quad (23)$$

where  $\otimes$  is set product.

This construction is described by the schema **U1\_C\_Parallel** (Section 26.5.13).

## 20.9. Sum

### Definition 20.19 (Sum for **PosL**)

For a list of maps  $\llbracket \ell_k : \text{kdom}_k \rightarrow_{\text{PosL}} \text{kcod}_k \rrbracket$  we define the sum as

$$\begin{aligned} \text{L1\_C\_Sum}(\llbracket \ell_k \rrbracket) : \text{P\_C\_Sum}(\llbracket \text{kdom}_k \rrbracket) &\rightarrow_{\text{PosL}} \text{P\_C\_Sum}(\llbracket \text{kcod}_k \rrbracket) \\ \langle k, r \rangle &\longmapsto \ell_k(r) \circ \downarrow \text{inj}_k \end{aligned} \quad (24)$$

where  $\text{inj}_k$  is the injection from  $\text{kcod}_k$  to  $\text{P\_C\_Sum}(\llbracket \text{kcod}_k \rrbracket)$ .

### Definition 20.20 (Sum for **PosU**)

For a list of maps  $\llbracket u_k : \text{kdom}_k \rightarrow_{\text{PosU}} \text{kcod}_k \rrbracket$  we define the sum as

$$\begin{aligned} \text{U1\_C\_Sum}(\llbracket u_k \rrbracket) : \text{P\_C\_Sum}(\llbracket \text{kdom}_k \rrbracket) &\rightarrow_{\text{PosU}} \text{P\_C\_Sum}(\llbracket \text{kcod}_k \rrbracket) \\ \langle k, f^* \rangle &\longmapsto u_k(f^*) \circ \uparrow \text{inj}_k \end{aligned} \quad (25)$$

where  $\text{inj}_k$  is the injection from  $\text{kcod}_k$  to  $\text{P\_C\_Sum}(\llbracket \text{kcod}_k \rrbracket)$ .

## 20.10. Codomain Sum

### Definition 20.21 (Codomain sum for **PosL**)

For a list of maps  $\llbracket \ell_k : \text{kdom} \rightarrow_{\text{PosL}} \text{kcod}_k \rrbracket$  with the same domain, the codomain sum of these maps is a map that combines the codomains of all maps:

$$\begin{aligned} \text{L1\_C\_CodSum}(\llbracket \ell_k \rrbracket) : \text{kdom} &\rightarrow_{\text{PosL}} \text{P\_C\_Sum}(\llbracket \text{kcod}_k \rrbracket) \\ r &\longmapsto \bigcup_k \ell_k(r) \circ \downarrow \text{inj}_k \end{aligned} \quad (26)$$

where  $\text{inj}_k$  is the injection from  $\text{kcod}_k$  to  $\text{P\_C\_Sum}(\llbracket \text{kcod}_k \rrbracket)$ .

This construction is described by the schema **L1\_C\_CodSum** (Section 26.4.10).

**Definition 20.22** (Codomain sum for **PosU**)

For a list of maps  $\llbracket u_k : \text{kdom} \rightarrow_{\text{PosU}} \text{kcod}_k \rrbracket$  with the same domain, the codomain sum of these maps is a map that combines the codomains of all maps:

$$\begin{aligned} \text{U1\_C\_CodSum}(\llbracket u_k \rrbracket) : \text{kdom} &\rightarrow_{\text{PosU}} \text{P\_C\_Sum}(\llbracket \text{kcod}_k \rrbracket) \\ f^* &\longmapsto \bigcup_k u_k(f^*) \uparrow \text{inj}_k \end{aligned} \quad (27)$$

where  $\text{inj}_k$  is the injection from  $\text{kcod}_k$  to  $\text{P\_C\_Sum}(\llbracket \text{kcod}_k \rrbracket)$ .

This construction is described by the schema **U1\_C\_CodSum** (Section 26.5.10).

## 20.11. Product of maps

**Definition 20.23** (Product for **PosL**)

For a list of maps  $\llbracket \ell_k : \text{kdom} \rightarrow_{\text{PosL}} \text{kcod}_k \rrbracket$ , we define:

$$\begin{aligned} \text{L1\_C\_Product}(\llbracket \ell_k \rrbracket) : \text{kdom} &\rightarrow_{\text{PosL}} \text{P\_C\_Product}(\llbracket \text{kcod}_k \rrbracket) \\ r &\longmapsto \otimes_k \ell_k(r) \end{aligned} \quad (28)$$

where  $\otimes$  is set product.

This construction is described by the schema **L1\_C\_Product** (Section 26.4.15).

**Definition 20.24** (Product for **PosU**)

For a list of maps  $\llbracket u_k : \text{kdom} \rightarrow_{\text{PosU}} \text{kcod}_k \rrbracket$ , we define:

$$\begin{aligned} \text{U1\_C\_Product}(\llbracket u_k \rrbracket) : \text{kdom} &\rightarrow_{\text{PosU}} \text{P\_C\_Product}(\llbracket \text{kcod}_k \rrbracket) \\ f^* &\longmapsto \otimes_k u_k(f^*) \end{aligned} \quad (29)$$

where  $\otimes$  is set product.

This construction is described by the schema **U1\_C\_Product** (Section 26.5.15).

## 20.12. Series composition

**Definition 20.25** (Binary series composition for **PosL**)

Given two maps  $\ell_1 : \text{kdom}_1 \rightarrow_{\text{PosL}} \text{kcod}_1$  and  $\ell_2 : \text{kdom}_2 \rightarrow_{\text{PosL}} \text{kcod}_2$  where  $\text{kcod}_1 \subseteq \text{kdom}_2$ , the series composition is:

$$\begin{aligned} (\ell_1 \circ \ell_2) : \text{kdom}_1 &\rightarrow_{\text{PosL}} \text{kcod}_2 \\ r &\longmapsto \bigcup_{y \in \ell_1(r)} \ell_2(y) \end{aligned} \quad (30)$$

This operation is associative and so the  $n$ -ary series composition  $\text{L1\_C\_Series}(\llbracket \ell_k \rrbracket)$  is defined for a family of maps  $\ell_k : \text{kdom}_k \rightarrow_{\text{PosL}} \text{kcod}_k$  whenever for  $k = 1, \dots, n-1$ ,  $\text{kcod}_k \subseteq \text{kdom}_{k+1}$ .

This construction is described by the schema **L1\_C\_Series** (Section 26.4.16).



**Definition 20.26** (Binary series composition for **PosU**)

Given two maps  $u_1 : \text{kdom}_1 \rightarrow_{\text{PosU}} \text{kcod}_1$  and  $u_2 : \text{kdom}_2 \rightarrow_{\text{PosU}} \text{kcod}_2$  where  $\text{kcod}_1 \subseteq \text{kdom}_2$ , the series composition is:

$$(u_1 \circ u_2) : \text{kdom}_1 \rightarrow_{\text{PosU}} \text{kcod}_2$$

$$f^* \longmapsto \bigcup_{y \in u_1(f^*)} u_2(y) \quad (31)$$

This operation is associative and so the  $n$ -ary series composition  $\text{U1\_C\_Series}(\llbracket u_k \rrbracket)$  is defined for a family of maps  $u_k : \text{kdom}_k \rightarrow_{\text{PosU}} \text{kcod}_k$  whenever for  $k = 1, \dots, n-1$ ,  $\text{kcod}_k \subseteq \text{kdom}_{k+1}$ .

This construction is described by the schema **U1\_C\_Series** (Section 26.5.16).

**20.13. Union and Intersection of maps****Definition 20.27** (Union for **PosL**)

For a family of maps  $\llbracket \ell_k : \text{kdom} \rightarrow_{\text{PosL}} \text{kcod} \rrbracket$ , we define:

$$\text{L1\_C\_Union}(\llbracket \ell_k \rrbracket) : \text{kdom} \rightarrow_{\text{PosL}} \text{kcod}$$

$$r \longmapsto \bigcup_k \ell_k(r) \quad (32)$$

This construction is described by the schema **L1\_C\_Union** (Section 26.4.18).

**Definition 20.28** (Union for **PosU**)

For a list of maps  $\llbracket u_k : \text{kdom} \rightarrow_{\text{PosU}} \text{kcod}_k \rrbracket$ , we define:

$$\text{U1\_C\_Union}(\llbracket u_k \rrbracket) : \text{kdom} \rightarrow_{\text{PosU}} \text{kcod}$$

$$f^* \longmapsto \bigcup_k u_k(f^*) \quad (33)$$

This construction is described by the schema **U1\_C\_Union** (Section 26.5.18).

**Definition 20.29** (Intersection for **PosL**)

For a list of maps  $\llbracket \ell_k : \text{kdom} \rightarrow_{\text{PosL}} \text{kcod}_k \rrbracket$ , we define:

$$\text{L1\_C\_Intersection}(\llbracket \ell_k \rrbracket) : \text{kdom} \rightarrow_{\text{PosL}} \text{kcod}$$

$$r \longmapsto \bigcap_k \ell_k(r) \quad (34)$$

This construction is described by the schema **L1\_C\_Intersection** (Section 26.4.17).

**Definition 20.30** (Intersection for **PosU**)

For a list of maps  $\llbracket u_k : \text{kdom} \rightarrow_{\text{PosU}} \text{kcod}_k \rrbracket$ , we define:

$$\text{U1\_C\_Intersection}(\llbracket u_k \rrbracket) : \text{kdom} \rightarrow_{\text{PosU}} \text{kcod}$$

$$f^* \longmapsto \bigcap_k u_k(f^*) \quad (35)$$

This construction is described by the schema **U1\_C\_Intersection** (Section 26.5.17).

**20.14. Trace**

**Definition 20.31** ( $L1\_C\_Trace$ )

Given a morphism

$$\ell_0 : P\_C\_Product(\llbracket kdom_1, kdom_2 \rrbracket) \rightarrow_{PosL} P\_C\_Product(\llbracket kcod_1, kcod_2 \rrbracket) \quad (36)$$

such that  $kcod_2 \subseteq kdom_2$  we define the morphism

$$\begin{aligned} L1\_C\_Trace(\ell_0) : kdom_1 &\rightarrow_{PosL} kcod_1 \\ r_1 &\longmapsto \downarrow \{f_1 \in kcod_1 \text{ such that } \exists f_2 \in kcod_2 : \langle f_1, f_2 \rangle \in \ell_0(r_1, f_2)\} \end{aligned} \quad (37)$$

*This construction is described by the schema  $L1\_C\_Trace$  (Section 26.4.20).*

**Definition 20.32** ( $U1\_C\_Trace$ )

Given a morphism

$$u_0 : P\_C\_Product(\llbracket kdom_1, kdom_2 \rrbracket) \rightarrow_{PosU} P\_C\_Product(\llbracket kcod_1, kcod_2 \rrbracket) \quad (38)$$

such that  $kcod_2 \subseteq kdom_2$  we define the morphism

$$\begin{aligned} U1\_C\_Trace(u_0) : kdom_1 &\rightarrow_{PosU} kcod_1 \\ f_1^* &\longmapsto \uparrow \{r_1 \in kcod_1 \text{ such that } \exists r_2 \in kcod_2 : \langle r_1, r_2 \rangle \in u_0(f_1^*, r_2)\} \end{aligned} \quad (39)$$

*This construction is described by the schema  $U1\_C\_Trace$  (Section 26.5.20).*

## 21. PosLI and PosUI catalog

### 21.1. Identity

**Definition 21.1**

Given a poset  $\mathbf{P}$ , we define

$$\begin{aligned} \text{L\_Identity}(\mathbf{P}) : \mathbf{P} &\rightarrow_{\text{PosLI}} \mathbf{P} \{\mathbf{1}\} \\ r &\longmapsto \downarrow \langle r, * \rangle \end{aligned} \quad (1)$$

This construction is described by the schema  $\text{L\_Identity}$  (Section 26.6.2).

Here,  $\mathbf{1}$  is the smash product of 0 posets,  $*$  is the unique element of  $\mathbf{1}$ .

**Definition 21.2**

Given a poset  $\mathbf{P}$ , we define

$$\begin{aligned} \text{U\_Identity}(\mathbf{P}) : \mathbf{P} &\rightarrow_{\text{PosUI}} \mathbf{P} \{\mathbf{1}\} \\ f^* &\longmapsto \uparrow \langle f^*, * \rangle \end{aligned} \quad (2)$$

This construction is described by the schema  $\text{U\_Identity}$  (Section 26.7.2).

### 21.2. Constant maps

**Definition 21.3**

Given a poset  $\mathbf{kdom}$ , a poset  $\mathbf{kcod}$ , a poset  $\mathbf{kimp}$  and an antichain  $\mathbf{A}$  of  $\text{P\_C\_Lexicographic}(\llbracket \mathbf{kcod}, \mathbf{kimp}^{\text{op}} \rrbracket)$ , we define the constant map

$$\begin{aligned} \text{L\_Constant}(\mathbf{kdom}, \mathbf{kcod}, \mathbf{kimp}, \mathbf{A}) : \mathbf{kdom} &\rightarrow_{\text{PosLI}} \mathbf{kcod} \{\mathbf{kimp}\} \\ r &\longmapsto \downarrow \mathbf{A} \end{aligned} \quad (3)$$

This construction is described by the schema  $\text{L\_Constant}$  (Section 26.6.1).

**Definition 21.4**

Given a poset  $\mathbf{kdom}$ , a poset  $\mathbf{kcod}$ , a poset  $\mathbf{kimp}$  and an antichain  $\mathbf{B}$  of  $\text{P\_C\_Lexicographic}(\llbracket \mathbf{kcod}, \mathbf{kimp} \rrbracket)$ , we define the constant map

$$\begin{aligned} \text{U\_Constant}(\mathbf{kdom}, \mathbf{kcod}, \mathbf{kimp}, \mathbf{B}) : \mathbf{kdom} &\rightarrow_{\text{PosUI}} \mathbf{kcod} \{\mathbf{kimp}\} \\ f^* &\longmapsto \uparrow \mathbf{B} \end{aligned} \quad (4)$$

This construction is described by the schema  $\text{U\_Constant}$  (Section 26.7.1).

### 21.3. Catalog maps

**Definition 21.5**

Given three posets  $\mathbf{F}, \mathbf{R}, \mathbf{I}$  and a list of options

$$\llbracket \langle f_k, r_k^*, i_k \rangle \rrbracket \subseteq \text{P\_C\_Product}(\llbracket \mathbf{F}, \mathbf{R}^{\text{op}}, \mathbf{I} \rrbracket) \quad (5)$$

we define the catalog map

$$\begin{aligned} \text{L\_Catalog}(\llbracket \langle f_k, r_k^*, i_k \rangle \rrbracket) : \mathbf{R} &\rightarrow_{\text{PosLI}} \mathbf{F}\{\mathbf{I}\} \\ r &\longmapsto \downarrow \bigcup_i \begin{cases} \{\langle f_k, i_k \rangle\} & \text{if } r_k^* \leq r, \\ \emptyset & \text{otherwise} \end{cases} \end{aligned} \quad (6)$$

This construction is described by the schema **L\_Catalog** (Section 26.6.4).

**Definition 21.6**

Given three posets  $\mathbf{F}, \mathbf{R}, \mathbf{I}$  and a list of options

$$\llbracket \langle f_k, r_k^*, i_k \rangle \rrbracket \subseteq \text{P\_C\_Product}(\llbracket \mathbf{F}, \mathbf{R}^{\text{op}}, \mathbf{I} \rrbracket) \quad (7)$$

we define the catalog map

$$\begin{aligned} \text{U\_Catalog}(\llbracket \langle f_k, r_k^*, i_k \rangle \rrbracket) : \mathbf{F} &\rightarrow_{\text{PosUI}} \mathbf{R}\{\mathbf{I}\} \\ f^* &\longmapsto \uparrow \bigcup_i \begin{cases} \{\langle r_k^*, i_k \rangle\} & \text{if } f^* \leq f_k, \\ \emptyset & \text{otherwise} \end{cases} \end{aligned} \quad (8)$$

This construction is described by the schema **U\_Catalog** (Section 26.7.4).

## 21.4. Lifting maps

**Definition 21.7**

Given a morphism

$$\ell : \text{kdom} \rightarrow_{\text{PosL}} \text{kcod} \quad (9)$$

and a monotone map

$$\text{tr} : \text{kcod} \times \text{kdom}^{\text{op}} \rightarrow_{\text{Pos}} \text{kimp} \quad (10)$$

we construct a new morphism

$$\begin{aligned} \text{L\_L\_Lift1\_Transform}(\ell, \text{tr}) : \text{kdom} &\rightarrow_{\text{PosLI}} \text{kcod} \{\text{kimp}\} \\ r &\longmapsto \downarrow \bigcup_{f \in \ell(r)} \{\langle f, \text{tr}(f, r) \rangle\} \end{aligned} \quad (11)$$

This construction is described by the schema **L\_L\_Lift1\_Transform** (Section 26.6.14).

**Definition 21.8**

Given a morphism

$$u : \text{kdom} \rightarrow_{\text{PosU}} \text{kcod} \quad (12)$$

and a monotone map

$$\text{tr} : \text{P\_C\_Product}(\llbracket \text{kcod}, \text{kdom}^{\text{op}} \rrbracket) \rightarrow_{\text{Pos}} \text{kimp} \quad (13)$$

we construct a new morphism

$$\begin{aligned} \text{U\_L\_Lift1\_Transform}(u, \text{tr}) : \text{kdom} &\rightarrow_{\text{PosUI}} \text{kcod} \{\text{kimp}\} \\ f^* &\longmapsto \uparrow \bigcup_{r \in u(f^*)} \{\langle r, \text{tr}(r, f^*) \rangle\} \end{aligned} \quad (14)$$

This construction is described by the schema **U\_L\_Lift1\_Transform** (Section 26.7.14).

**Definition 21.9**

Given a morphism

$$\ell : \text{kdom} \rightarrow_{\text{PosL}} \text{kcod}, \quad (15)$$

a poset  $\text{kimp}$  and a value  $i_0 \in \text{kimp}$  we construct a new morphism  $\text{L\_L\_Lift1\_Constant}(\ell, \text{kimp}, i_0)$  as a particular case of **L\_L\_Lift1\_Transform**

where  $g$  is the constant map  $g(y, x) = i_0$ .

This construction is described by the schema  $\text{L\_L\_Lift1\_Constant}$  (Section 26.6.13).

**Definition 21.10**  
Given a morphism

$$u : \text{ldom} \rightarrow_{\text{PosU}} \text{kcod}, \quad (16)$$

a poset  $\text{kimp}$  and a value  $i_0 \in \text{kimp}$  we construct a new morphism  $\text{U\_L\_Lift1\_Constant}(u, \text{kimp}, i_0)$  as a particular case of  $\text{U\_L\_Lift1\_Transform}$  where  $g$  is the constant map  $g(y, x) = i_0$ .

This construction is described by the schema  $\text{U\_L\_Lift1\_Constant}$  (Section 26.7.13).

## 21.5. Series composition

**Definition 21.11** (Binary series composition)  
Given two morphisms

$$\ell_1 : \text{ldom}_1 \rightarrow_{\text{PosLI}} \text{kcod}_1 \{\text{kimp}_1\} \quad \text{and} \quad \ell_2 : \text{ldom}_2 \rightarrow_{\text{PosLI}} \text{kcod}_2 \{\text{kimp}_2\} \quad (17)$$

such that  $\text{kcod}_1 \subseteq \text{ldom}_2$ , the series composition is the morphism

$$\begin{aligned} \text{L\_C\_Series}(\llbracket \ell_1, \ell_2 \rrbracket) : \text{ldom}_1 &\rightarrow_{\text{PosLI}} \text{kcod}_2 \{\text{P\_C\_ProductSmash}(\llbracket \text{kimp}_1, \text{kimp}_2 \rrbracket)\} \\ r &\longmapsto \downarrow \bigcup_{\langle x_1, i_1 \rangle \in \ell_1(r)} \{ \langle x_2, [i_1 \mid i_2] \rangle \mid \langle x_2, i_2 \rangle \in \ell_2(x_1) \} \end{aligned} \quad (18)$$

This composition is associative, therefore we can define  $\text{L\_C\_Series}(\llbracket \ell_i \rrbracket)$  whenever we have a list of morphisms  $\llbracket \ell_i \rrbracket$  such that  $\text{kcod}_i \subseteq \text{ldom}_{i+1}$  for all  $i = 1, \dots, n-1$ .

This construction is described by the schema  $\text{L\_C\_Series}$  (Section 26.6.6).

We repeat the same construction for the series composition in  $\text{PosUI}$ .

**Definition 21.12** (Binary series composition)  
Given two morphisms

$$u_1 : \text{ldom}_1 \rightarrow_{\text{PosUI}} \text{kcod}_1 \{\text{kimp}_1\} \quad \text{and} \quad u_2 : \text{ldom}_2 \rightarrow_{\text{PosUI}} \text{kcod}_2 \{\text{kimp}_2\} \quad (19)$$

such that  $\text{kcod}_1 \subseteq \text{ldom}_2$ , the series composition is the morphism

$$\begin{aligned} \text{U\_C\_Series}(\llbracket u_1, u_2 \rrbracket) : \text{ldom}_1 &\rightarrow_{\text{PosUI}} \text{kcod}_2 \{\text{P\_C\_ProductSmash}(\llbracket \text{kimp}_1, \text{kimp}_2 \rrbracket)\} \\ f^* &\longmapsto \uparrow \bigcup_{\langle x_1, i_1 \rangle \in u_1(f^*)} \{ \langle x_2, [i_1 \mid i_2] \rangle \mid \langle x_2, i_2 \rangle \in u_2(x_1) \} \end{aligned} \quad (20)$$

This composition is associative, therefore we can define  $\text{U\_C\_Series}(\llbracket u_i \rrbracket)$  whenever we have a list of morphisms  $\llbracket u_i \rrbracket$  such that  $\text{kcod}_i \subseteq \text{ldom}_{i+1}$  for all  $i = 1, \dots, n-1$ .

This construction is described by the schema  $\text{U\_C\_Series}$  (Section 26.7.6).

## 21.6. Parallel composition

**Definition 21.13** (Parallel composition in  $\text{PosLI}$ )

Given a list of  $n$  morphisms  $\llbracket \ell_k : \text{ldom}_k \rightarrow_{\text{PosLI}} \text{kcod}_k \{\text{kimp}_k\} \rrbracket$  we define the parallel composition as the morphism

$$\begin{aligned} \text{L\_C\_Parallel}(\llbracket \ell_k \rrbracket) : \text{P\_C\_Product}(\llbracket \text{ldom}_k \rrbracket) &\rightarrow_{\text{PosUI}} \text{P\_C\_Product}(\llbracket \text{kcod}_k \rrbracket) \{\text{P\_C\_ProductSmash}(\llbracket \text{kimp}_k \rrbracket)\} \\ \langle r_1, \dots, r_n \rangle &\longmapsto \downarrow \bigcup_{\substack{\langle y_k, i_k \rangle \in \ell_k(r_k) \\ k=1:n \text{ times}}} \{ \langle \langle y_1, \dots, y_n \rangle, [i_1 \mid \dots \mid i_n] \rangle \} \end{aligned} \quad (21)$$

This construction is described by the schema  $\text{L\_C\_Parallel}$  (Section 26.6.5).

**Definition 21.14** (Parallel composition in **PosUI**)

Given a list of  $n$  morphisms  $\llbracket u_k : \mathbf{kdom}_k \rightarrow_{\mathbf{PosUI}} \mathbf{kcod}_k \{ \mathbf{kimp}_k \} \rrbracket$  we define the parallel composition as the morphism

$$\begin{aligned} \mathbf{U\_C\_Parallel}(\llbracket u_k \rrbracket) : \mathbf{P\_C\_Product}(\llbracket \mathbf{kdom}_k \rrbracket) &\rightarrow_{\mathbf{PosUI}} \mathbf{P\_C\_Product}(\llbracket \mathbf{kdom}_k \rrbracket) \{ \mathbf{P\_C\_ProductSmash}(\llbracket \mathbf{kimp}_k \rrbracket) \} \\ \langle f_1^*, \dots, f_n^* \rangle &\longmapsto \uparrow \underbrace{\bigcup_{k=1:n \text{ times}} \bigcup_{\langle r_k, i_k \rangle \in u_k(f_k^*)} \{ \langle \langle r_1, \dots, r_n \rangle, [i_1 \mid \dots \mid i_n] \rangle \}}_{k=1:n \text{ times}} \end{aligned} \quad (22)$$

This construction is described by the schema **U\_C\_Parallel** (Section 26.7.5).

## 21.7. Intersection of maps

**Definition 21.15** (Intersection of maps)

Given a poset  $\mathbf{kdom}$ , a meet semilattice  $\mathbf{kcod}$ , and a list of  $n$  morphisms  $\llbracket \ell_k : \mathbf{kdom} \rightarrow_{\mathbf{PosLI}} \mathbf{kcod} \{ \mathbf{kimp}_k \} \rrbracket$  we define the intersection as the morphism

$$\begin{aligned} \mathbf{L\_C\_Intersection}(\llbracket \ell_k \rrbracket) : \mathbf{kdom} &\rightarrow_{\mathbf{PosLI}} \mathbf{kcod} \{ \mathbf{P\_C\_ProductSmash}(\llbracket \mathbf{kimp}_k \rrbracket) \} \\ \langle r_1, \dots, r_n \rangle &\longmapsto \downarrow \underbrace{\bigcup_{k=1:n \text{ times}} \bigcup_{\langle y_k, i_k \rangle \in \ell_k(r_k)} \{ \langle \wedge_{\mathbf{kcod}}^n (y_1, \dots, y_n), [i_1 \mid \dots \mid i_n] \rangle \}}_{k=1:n \text{ times}} \end{aligned} \quad (23)$$

where  $\wedge_{\mathbf{kcod}}^n$  is the meet of  $n$  elements in  $\mathbf{kcod}$ .

This construction is described by the schema **L\_C\_Intersection** (Section 26.6.7).

**Definition 21.16** (Intersection of maps)

Given a poset  $\mathbf{kdom}$ , a join semilattice  $\mathbf{kcod}$ , and a list of  $n$  morphisms  $\llbracket u_k : \mathbf{kdom} \rightarrow_{\mathbf{PosUI}} \mathbf{kcod} \{ \mathbf{kimp}_k \} \rrbracket$  we define the intersection as the morphism

$$\begin{aligned} \mathbf{U\_C\_Intersection}(\llbracket u_k \rrbracket) : \mathbf{kdom} &\rightarrow_{\mathbf{PosUI}} \mathbf{kcod} \{ \mathbf{P\_C\_ProductSmash}(\llbracket \mathbf{kimp}_k \rrbracket) \} \\ \langle f_1^*, \dots, f_n^* \rangle &\longmapsto \uparrow \underbrace{\bigcup_{k=1:n \text{ times}} \bigcup_{\langle r_k, i_k \rangle \in u_k(f_k^*)} \{ \langle \vee_{\mathbf{kcod}}^n (r_1, \dots, r_n), [i_1 \mid \dots \mid i_n] \rangle \}}_{k=1:n \text{ times}} \end{aligned} \quad (24)$$

where  $\vee_{\mathbf{kcod}}^n$  is the join of  $n$  elements in  $\mathbf{kcod}$ .

This construction is described by the schema **U\_C\_Intersection** (Section 26.7.7).

## 21.8. Union of maps

**Definition 21.17** (Union of maps)

Given a poset  $\mathbf{kdom}$ , a meet semilattice  $\mathbf{kcod}$ , and a list of  $n$  morphisms  $\llbracket \ell_k : \mathbf{kdom} \rightarrow_{\mathbf{PosLI}} \mathbf{kcod} \{ \mathbf{kimp}_k \} \rrbracket$  we define the union as the morphism

$$\begin{aligned} \mathbf{L\_C\_Union}(\llbracket \ell_k \rrbracket) : \mathbf{kdom} &\rightarrow_{\mathbf{PosLI}} \mathbf{kcod} \{ \mathbf{P\_C\_SumSmash}(\llbracket \mathbf{kimp}_k \rrbracket) \} \\ \langle r_1, \dots, r_n \rangle &\longmapsto \downarrow \bigcup_k \bigcup_{\langle y, i \rangle \in \ell_k(r_k)} \langle y, \text{inj}_k(i) \rangle \end{aligned} \quad (25)$$

where  $\text{inj}_k$  is the injection from  $\mathbf{kimp}_k$  to  $\mathbf{P\_C\_SumSmash}(\llbracket \mathbf{kimp}_k \rrbracket)$ .

This construction is described by the schema **L\_C\_Union** (Section 26.6.8).

**Definition 21.18** (Union of maps)

Given a poset  $\mathbf{kdom}$ , a join semilattice  $\mathbf{kcod}$ , and a list of  $n$  morphisms  $\llbracket u_k : \mathbf{kdom} \rightarrow_{\mathbf{PosUI}} \mathbf{kcod} \{\mathbf{kimp}_k\} \rrbracket$  we define the morphism

$$\begin{aligned} \mathbf{U\_C\_Union}(\llbracket u_k \rrbracket) : \mathbf{kdom} &\rightarrow_{\mathbf{PosUI}} \mathbf{kcod} \{\mathbf{P\_C\_SumSmash}(\llbracket \mathbf{kimp}_k \rrbracket)\} \\ \langle f_1^*, \dots, f_n^* \rangle &\longmapsto \biguparrow_k \bigcup_{\langle r, i \rangle \in u_k(f^*)} \langle r, \mathbf{inj}_k(i) \rangle \end{aligned} \quad (26)$$

where  $\mathbf{inj}_k$  is the injection from  $\mathbf{kimp}_k$  to  $\mathbf{P\_C\_SumSmash}(\llbracket \mathbf{kimp}_k \rrbracket)$ .

This construction is described by the schema  $\mathbf{U\_C\_Union}$  (Section 26.7.8).

## 21.9. Transforming maps

**Definition 21.19**

Given a morphism

$$\ell_0 : \mathbf{kdom} \rightarrow_{\mathbf{PosLI}} \mathbf{kcod} \{\mathbf{kimp}\} \quad (27)$$

and a monotone map

$$g : \mathbf{kimp} \rightarrow_{\mathbf{Pos}} \mathbf{kimp}' \quad (28)$$

we construct a new morphism

$$\begin{aligned} \mathbf{L\_C\_ITransform}(\ell_0, g) : \mathbf{kdom} &\rightarrow_{\mathbf{PosLI}} \mathbf{kcod} \{\mathbf{kimp}'\} \\ r &\longmapsto \biguparrow_{\langle f, i \rangle \in \ell_0(r)} \{ \langle f, g(i) \rangle \} \end{aligned} \quad (29)$$

This construction is described by the schema  $\mathbf{L\_C\_ITransform}$  (Section 26.6.9).

**Definition 21.20**

Given a morphism

$$u_0 : \mathbf{kdom} \rightarrow_{\mathbf{PosUI}} \mathbf{kcod} \{\mathbf{kimp}\} \quad (30)$$

and a monotone map

$$g : \mathbf{kimp} \rightarrow_{\mathbf{Pos}} \mathbf{kimp}' \quad (31)$$

we construct a new morphism

$$\begin{aligned} \mathbf{U\_C\_ITransform}(u_0, g) : \mathbf{kdom} &\rightarrow_{\mathbf{PosUI}} \mathbf{kcod} \{\mathbf{kimp}'\} \\ f^* &\longmapsto \biguparrow_{\langle r, i \rangle \in u_0(f^*)} \{ \langle r, g(i) \rangle \} \end{aligned} \quad (32)$$

This construction is described by the schema  $\mathbf{U\_C\_ITransform}$  (Section 26.7.9).

## 21.10. Trace

**Definition 21.21**

Given a morphism

$$\ell_0 : \mathbf{P\_C\_Product}(\llbracket \mathbf{kdom}_1, \mathbf{kdom}_2 \rrbracket) \rightarrow_{\mathbf{PosLI}} \mathbf{P\_C\_Product}(\llbracket \mathbf{kcod}_1, \mathbf{kcod}_2 \rrbracket) \{\mathbf{kimp}\} \quad (33)$$

such that  $\mathbf{kcod}_2 \subseteq \mathbf{kdom}_2$  we define the morphism

$$\begin{aligned} \mathbf{L\_C\_Trace}(\ell_0) : \mathbf{kdom}_1 &\rightarrow_{\mathbf{PosLI}} \mathbf{kcod}_1 \{\mathbf{kimp}\} \\ r_1 &\longmapsto \biguparrow \{ \langle f_1, i \rangle \text{ such that } \exists f_2 \in \mathbf{kcod}_2 : \langle \langle f_1, f_2 \rangle, i \rangle \in \ell_0(r_1, f_2) \} \end{aligned} \quad (34)$$

This construction is described by the schema  $\mathbf{L\_C\_Trace}$  (Section 26.6.11).

**Definition 21.22**

Given a morphism

$$u_0 : \text{P\_C\_Product}(\llbracket \text{kd}_{\text{om}_1}, \text{kd}_{\text{om}_2} \rrbracket) \rightarrow_{\text{PosUI}} \text{P\_C\_Product}(\llbracket \text{kc}_{\text{od}_1}, \text{kc}_{\text{od}_2} \rrbracket) \{\text{kip}\} \quad (35)$$

such that  $\text{kc}_{\text{od}_2} \subseteq \text{kd}_{\text{om}_2}$  we define the morphism

$$\begin{aligned} \text{U\_C\_Trace}(u_0) : \text{kd}_{\text{om}_1} &\rightarrow_{\text{PosUI}} \text{kc}_{\text{od}_1} \{\text{kip}\} \\ f_1^* &\longmapsto \uparrow \{ \langle r_1, i \rangle \text{ such that } \exists r_2 \in \text{kc}_{\text{od}_2} : \langle \langle r_1, r_2 \rangle, i \rangle \in u_0(f_1^*, r_2) \} \end{aligned} \quad (36)$$

This construction is described by the schema **U\_C\_Trace** (Section 26.7.11).



## 22. SPos $\mathbf{L}$ and SPos $\mathbf{U}$ catalog

### 22.1. Identities

The identity morphism for SPos $\mathbf{L}$  can be defined as

$$\text{SL1\_Identity}(\mathbf{P}) \doteq \text{SL1\_Exact}(\text{L1\_Identity}(\mathbf{P})) \quad (1)$$

*This construction is described by the schema SL1\\_Identity (Section 26.8.3).*

Likewise the identity morphism for SPos $\mathbf{U}$  can be defined as

$$\text{SU1\_Identity}(\mathbf{P}) \doteq \text{SU1\_Exact}(\text{U1\_Identity}(\mathbf{P})) \quad (2)$$

*This construction is described by the schema SU1\\_Identity (Section 26.9.3).*

### 22.2. Lifting

**Definition 22.1** (Lift of a Pos $\mathbf{L}$  to a SPos $\mathbf{L}$ )

Given a Pos $\mathbf{L}$  morphism

$$\ell : \text{ldom} \rightarrow_{\text{PosLI}} \text{kcod}, \quad (3)$$

we can lift it as a morphism of SPos $\mathbf{L}$

$$\text{SL1\_Exact}(\ell) : \{\mathbf{S}\} \text{ldom} \rightarrow_{\text{SPosL}} \text{kcod} \quad (4)$$

by setting

$$\mathbf{S}^{\ominus} = \mathbf{S}^{\ominus} = \mathbf{1} \quad (5)$$

and

$$\text{sl}^{\ominus} : * \mapsto \ell \quad (6)$$

$$\text{sl}^{\ominus} : * \mapsto \ell \quad (7)$$

*This construction is described by the schema SL1\\_Exact (Section 26.8.14).*

**Definition 22.2** (Lift of a Pos $\mathbf{U}$  to a SPos $\mathbf{U}$ )

Given a Pos $\mathbf{U}$  morphism

$$u : \text{ldom} \rightarrow_{\text{PosUI}} \text{kcod}, \quad (8)$$

we can lift it as a morphism of SPos $\mathbf{U}$

$$\text{SU1\_Exact}(u) : \{\mathbf{S}\} \text{ldom} \rightarrow_{\text{SPosU}} \text{kcod} \quad (9)$$

by setting

$$\mathbf{S}^{\ominus} = \mathbf{S}^{\ominus} = \mathbf{1} \quad (10)$$

and

$$\text{su}^{\ominus} : * \mapsto u \quad (11)$$

$$\text{su}^{\ominus} : * \mapsto u \quad (12)$$

*This construction is described by the schema SU1\\_Exact (Section 26.9.14).*

## 22.3. Parallel composition

### Definition 22.3

Given a list of  $n$  morphisms

$$[sl_k : \{S_k\} \text{ kdom}_k \rightarrow_{\text{SPosL}} \text{kcod}_k] \quad (13)$$

the parallel composition is

$$\text{SL1\_C\_Parallel}([sl_k]) : \{S\} \text{ P\_C\_Product}([kdom_k]) \rightarrow_{\text{SPosL}} \text{P\_C\_Product}([kcod_k]) \quad (14)$$

with

$$S^\ominus = \text{P\_C\_ProductSmash}([S_k^\ominus]) \quad S^\ominus = \text{P\_C\_ProductSmash}([S_k^\ominus]) \quad (15)$$

given by

$$sl^\ominus : [o_1 | \dots | o_n] \mapsto \text{L1\_C\_Parallel}([sl_k^\ominus(o_k)]) \quad (16)$$

$$sl^\ominus : [p_1 | \dots | p_n] \mapsto \text{L1\_C\_Parallel}([sl_k^\ominus(p_k)]) \quad (17)$$

This construction is described by the schema **SL1\_C\_Parallel** (Section 26.8.1).

### Definition 22.4

Given a list of  $n$  morphisms

$$[su_k : \{S_k\} \text{ kdom}_k \rightarrow_{\text{SPosU}} \text{kcod}_k] \quad (18)$$

the parallel composition is

$$\text{SU1\_C\_Parallel}([su_k]) : \{S\} \text{ P\_C\_Product}([kdom_k]) \rightarrow_{\text{SPosU}} \text{P\_C\_Product}([kcod_k]) \quad (19)$$

with

$$S^\ominus = \text{P\_C\_ProductSmash}([S_k^\ominus]) \quad S^\ominus = \text{P\_C\_ProductSmash}([S_k^\ominus]) \quad (20)$$

given by

$$su^\ominus : [o_1 | \dots | o_n] \mapsto \text{U1\_C\_Parallel}([su_k^\ominus(o_k)]) \quad (21)$$

$$su^\ominus : [p_1 | \dots | p_n] \mapsto \text{U1\_C\_Parallel}([su_k^\ominus(p_k)]) \quad (22)$$

This construction is described by the schema **SU1\_C\_Parallel** (Section 26.9.1).

## 22.4. Series composition

### Definition 22.5

Given a list of  $n$  morphisms

$$[sl_k : \{S_k\} \text{ kdom}_k \rightarrow_{\text{SPosL}} \text{kcod}_k] \quad (23)$$

such that for all  $k = 1, \dots, n-1$ , we have  $\text{kcod}_k \subseteq \text{kdom}_{k+1}$ , the series composition is

$$\text{SL1\_C\_Series}([sl_k]) : \{S\} \text{ kdom}_1 \rightarrow_{\text{SPosL}} \text{kcod}_n \quad (24)$$

with

$$S^\ominus = \text{P\_C\_ProductSmash}([S_k^\ominus]) \quad S^\ominus = \text{P\_C\_ProductSmash}([S_k^\ominus]) \quad (25)$$

given by

$$sl^\ominus : [o_1 | \dots | o_n] \mapsto \text{L1\_C\_Series}([sl_k^\ominus(o_k)]) \quad (26)$$

$$sl^\ominus : [p_1 | \dots | p_n] \mapsto \text{L1\_C\_Series}([sl_k^\ominus(p_k)]) \quad (27)$$

This construction is described by the schema **SL1\_C\_Series** (Section 26.8.2).

**Definition 22.6**

Given a list of  $n$  morphisms

$$\llbracket \text{su}_k : \{S_k\} \text{ kdom}_k \rightarrow_{\text{SPosU}} \text{ kcod}_k \rrbracket \quad (28)$$

the series composition is

$$\text{SU1\_C\_Series}(\llbracket \text{su}_k \rrbracket) : \{S\} \text{ kdom} \rightarrow_{\text{SPosU}} \text{ kcod} \quad (29)$$

with

$$S^{\ominus} = \text{P\_C\_ProductSmash}(\llbracket S_k^{\ominus} \rrbracket) \quad S^{\ominus} = \text{P\_C\_ProductSmash}(\llbracket S_k^{\ominus} \rrbracket) \quad (30)$$

given by

$$\text{su}^{\ominus} : [o_1 \mid \cdots \mid o_n] \mapsto \text{U1\_C\_Series}(\llbracket \text{su}_k^{\ominus}(o_k) \rrbracket) \quad (31)$$

$$\text{su}^{\ominus} : [p_1 \mid \cdots \mid p_n] \mapsto \text{U1\_C\_Series}(\llbracket \text{su}_k^{\ominus}(p_k) \rrbracket) \quad (32)$$

*This construction is described by the schema **SU1\_C\_Series** (Section 26.9.2).*

## 22.5. Union

**Definition 22.7**

Given two posets  $\text{kdom}$  and  $\text{kcod}$  and a list of maps

$$\llbracket \text{sl}_k : \{S_k\} \text{ kdom} \rightarrow_{\text{SPosL}} \text{ kcod} \rrbracket \quad (33)$$

the union composition is

$$\text{SL1\_C\_Union}(\llbracket \text{sl}_k \rrbracket) : \{S\} \text{ kdom} \rightarrow_{\text{SPosL}} \text{ kcod} \quad (34)$$

with

$$S^{\ominus} = \text{P\_C\_ProductSmash}(\llbracket S_k^{\ominus} \rrbracket) \quad S^{\ominus} = \text{P\_C\_ProductSmash}(\llbracket S_k^{\ominus} \rrbracket) \quad (35)$$

given by

$$\text{sl}^{\ominus} : [o_1 \mid \cdots \mid o_n] \mapsto \text{L1\_C\_Union}(\llbracket \text{sl}_k^{\ominus}(o_k) \rrbracket) \quad (36)$$

$$\text{sl}^{\ominus} : [p_1 \mid \cdots \mid p_n] \mapsto \text{L1\_C\_Union}(\llbracket \text{sl}_k^{\ominus}(p_k) \rrbracket) \quad (37)$$

*This construction is described by the schema **SL1\_C\_Union** (Section 26.8.10).*

**Definition 22.8**

Given two posets  $\text{kdom}$  and  $\text{kcod}$  and a list of maps

$$\llbracket \text{su}_k : \{S_k\} \text{ kdom} \rightarrow_{\text{SPosU}} \text{ kcod} \rrbracket \quad (38)$$

the union composition is the morphism

$$\text{SU1\_C\_Union}(\llbracket \text{su}_k \rrbracket) : \{S\} \text{ kdom} \rightarrow_{\text{SPosU}} \text{ kcod} \quad (39)$$

with

$$S^{\ominus} = \text{P\_C\_ProductSmash}(\llbracket S_k^{\ominus} \rrbracket) \quad S^{\ominus} = \text{P\_C\_ProductSmash}(\llbracket S_k^{\ominus} \rrbracket) \quad (40)$$

given by

$$\text{su}^{\ominus} : [o_1 \mid \cdots \mid o_n] \mapsto \text{U1\_C\_Union}(\llbracket \text{su}_k^{\ominus}(o_k) \rrbracket) \quad (41)$$

$$\text{su}^{\ominus} : [p_1 \mid \cdots \mid p_n] \mapsto \text{U1\_C\_Union}(\llbracket \text{su}_k^{\ominus}(p_k) \rrbracket) \quad (42)$$

*This construction is described by the schema **SU1\_C\_Union** (Section 26.9.10).*

## 22.6. Intersection

### Definition 22.9

Given two posets  $\text{kdom}$  and  $\text{kcod}$  and a list of  $n$  morphisms

$$\llbracket \text{sl}_k : \{S_k\} \text{kdom} \rightarrow_{\text{SPosL}} \text{kcod} \rrbracket \quad (43)$$

the intersection composition is

$$\text{SL1\_C\_Intersection}(\llbracket \text{sl}_k \rrbracket) : \{S\} \text{kdom} \rightarrow_{\text{SPosL}} \text{kcod} \quad (44)$$

with

$$S^\ominus = \text{P\_C\_ProductSmash}(\llbracket S^\ominus_k \rrbracket) \quad S^\ominus = \text{P\_C\_ProductSmash}(\llbracket S^\ominus_k \rrbracket) \quad (45)$$

given by

$$\text{sl}^\ominus : [o_1 \mid \dots \mid o_n] \mapsto \text{L1\_C\_Intersection}(\llbracket \text{sl}_k^\ominus(o_k) \rrbracket) \quad (46)$$

$$\text{sl}^\ominus : [p_1 \mid \dots \mid p_n] \mapsto \text{L1\_C\_Intersection}(\llbracket \text{sl}_k^\ominus(p_k) \rrbracket) \quad (47)$$

*This construction is described by the schema  $\text{SL1\_C\_Intersection}$  (Section 26.8.9).*

### Definition 22.10

Given two posets  $\text{kdom}$  and  $\text{kcod}$  and a list of  $n$  morphisms

$$\llbracket \text{su}_k : \{S_k\} \text{kdom} \rightarrow_{\text{SPosU}} \text{kcod} \rrbracket \quad (48)$$

the intersection composition is

$$\text{SU1\_C\_Intersection}(\llbracket \text{su}_k \rrbracket) : \{S\} \text{kdom} \rightarrow_{\text{SPosU}} \text{kcod} \quad (49)$$

with

$$S^\ominus = \text{P\_C\_ProductSmash}(\llbracket S^\ominus_k \rrbracket) \quad S^\ominus = \text{P\_C\_ProductSmash}(\llbracket S^\ominus_k \rrbracket) \quad (50)$$

given by

$$\text{su}^\ominus : [o_1 \mid \dots \mid o_n] \mapsto \text{U1\_C\_Intersection}(\llbracket \text{su}_k^\ominus(o_k) \rrbracket) \quad (51)$$

$$\text{su}^\ominus : [p_1 \mid \dots \mid p_n] \mapsto \text{U1\_C\_Intersection}(\llbracket \text{su}_k^\ominus(p_k) \rrbracket) \quad (52)$$

*This construction is described by the schema  $\text{SU1\_C\_Intersection}$  (Section 26.9.9).*

## 22.7. Trace

### Definition 22.11

Given a morphism

$$\text{sl}_0 : \{S\} \text{P\_C\_Product}(\llbracket \text{kdom}_1, \text{kdom}_2 \rrbracket) \rightarrow_{\text{SPosL}} \text{P\_C\_Product}(\llbracket \text{kcod}_1, \text{kcod}_2 \rrbracket) \quad (53)$$

such that  $\text{kcod}_2 \subseteq \text{kdom}_2$  we define the morphism

$$\text{SL1\_C\_Trace}(\text{sl}) : \{S\} \text{kdom}_1 \rightarrow_{\text{SPosL}} \text{kcod}_1 \quad (54)$$

given by

$$\text{sl}^\ominus : o \mapsto \text{L1\_C\_Trace}(\text{sl}_0^\ominus(o)) \quad (55)$$

$$\text{sl}^\ominus : p \mapsto \text{L1\_C\_Trace}(\text{sl}_0^\ominus(p)) \quad (56)$$

*This construction is described by the schema  $\text{SL1\_C\_Trace}$  (Section 26.8.12).*

### Definition 22.12

Given a morphism

$$\text{su}_0 : \{S\} \text{P\_C\_Product}(\llbracket \text{kdom}_1, \text{kdom}_2 \rrbracket) \rightarrow_{\text{SPosU}} \text{P\_C\_Product}(\llbracket \text{kcod}_1, \text{kcod}_2 \rrbracket) \quad (57)$$

such that  $\text{kcod}_2 \subseteq \text{kdom}_2$  we define the morphism

$$\text{SU1\_C\_Trace}(\text{su}) : \{S\} \text{kdom}_1 \rightarrow_{\text{SPosU}} \text{kcod}_1 \quad (58)$$

given by

$$\text{su}^\ominus : o \mapsto \text{U1\_C\_Trace}(\text{su}_0^\ominus(o)) \quad (59)$$

$$\text{su}^\ominus : p \mapsto \text{U1\_C\_Trace}(\text{su}_0^\ominus(p)) \quad (60)$$

This construction is described by the schema **SU1\_C\_Trace** (Section 26.9.12).

## 22.8. Product

### Definition 22.13

Given a list of  $n$  morphisms

$$\llbracket \text{sl}_k : \{S_k\} \text{kdom} \rightarrow_{\text{SPosL}} \text{kcod}_k \rrbracket \quad (61)$$

the parallel composition is

$$\text{SL1\_C\_Product}(\llbracket \text{sl}_k \rrbracket) : \{S\} \text{kdom} \rightarrow_{\text{SPosL}} \text{P\_C\_Product}(\llbracket \text{kcod}_k \rrbracket) \quad (62)$$

with

$$S^\ominus = \text{P\_C\_ProductSmash}(\llbracket S^\ominus_k \rrbracket) \quad S^\ominus = \text{P\_C\_ProductSmash}(\llbracket S^\ominus_k \rrbracket) \quad (63)$$

given by

$$\text{sl}^\ominus : [o_1 \mid \dots \mid o_n] \mapsto \text{L1\_C\_Product}(\llbracket \text{sl}_k^\ominus(o_k) \rrbracket) \quad (64)$$

$$\text{sl}^\ominus : [p_1 \mid \dots \mid p_n] \mapsto \text{L1\_C\_Product}(\llbracket \text{sl}_k^\ominus(p_k) \rrbracket) \quad (65)$$

This construction is described by the schema **SL1\_C\_Product** (Section 26.8.8).

### Definition 22.14

Given a list of  $n$  morphisms

$$\llbracket \text{su}_k : \{S_k\} \text{kdom} \rightarrow_{\text{SPosU}} \text{kcod}_k \rrbracket \quad (66)$$

the parallel composition is

$$\text{SU1\_C\_Product}(\llbracket \text{su}_k \rrbracket) : \{S\} \text{kdom} \rightarrow_{\text{SPosU}} \text{P\_C\_Product}(\llbracket \text{kcod}_k \rrbracket) \quad (67)$$

with

$$S^\ominus = \text{P\_C\_ProductSmash}(\llbracket S^\ominus_k \rrbracket) \quad S^\ominus = \text{P\_C\_ProductSmash}(\llbracket S^\ominus_k \rrbracket) \quad (68)$$

given by

$$\text{su}^\ominus : [o_1 \mid \dots \mid o_n] \mapsto \text{U1\_C\_Product}(\llbracket \text{su}_k^\ominus(o_k) \rrbracket) \quad (69)$$

$$\text{su}^\ominus : [p_1 \mid \dots \mid p_n] \mapsto \text{U1\_C\_Product}(\llbracket \text{su}_k^\ominus(p_k) \rrbracket) \quad (70)$$

This construction is described by the schema **SU1\_C\_Product** (Section 26.9.8).

## 22.9. Sum

### Definition 22.15

Given a list of  $n$  morphisms

$$\llbracket \text{sl}_k : \{S_k\} \text{kdom} \rightarrow_{\text{SPosL}} \text{kcod}_k \rrbracket \quad (71)$$

the parallel composition is

$$\text{SL1\_C\_CodSum}(\llbracket \text{sl}_k \rrbracket) : \{S\} \text{kdom} \rightarrow_{\text{SPosL}} \text{P\_C\_Sum}(\llbracket \text{kcod}_k \rrbracket) \quad (72)$$

with

$$S^\ominus = \text{P\_C\_ProductSmash}(\llbracket S^\ominus_k \rrbracket) \quad S^\ominus = \text{P\_C\_ProductSmash}(\llbracket S^\ominus_k \rrbracket) \quad (73)$$

given by

$$\text{sl}^\oplus : [o_1 \mid \cdots \mid o_n] \mapsto \text{L1\_C\_CodSum}(\llbracket \text{sl}_k^\oplus(o_k) \rrbracket) \quad (74)$$

$$\text{sl}^\ominus : [p_1 \mid \cdots \mid p_n] \mapsto \text{L1\_C\_CodSum}(\llbracket \text{sl}_k^\ominus(p_k) \rrbracket) \quad (75)$$

This construction is described by the schema [SL1\\_C\\_CodSum](#) (Section 26.8.5).

**Definition 22.16**

Given a list of  $n$  morphisms

$$\llbracket \text{su}_k : \{S_k\} \text{ kdom} \rightarrow_{\text{SPosU}} \text{kcod}_k \rrbracket \quad (76)$$

the parallel composition is

$$\text{SU1\_C\_CodSum}(\llbracket \text{su}_k \rrbracket) : \{S\} \text{ kdom} \rightarrow_{\text{SPosU}} \text{P\_C\_Sum}(\llbracket \text{kcod}_k \rrbracket) \quad (77)$$

with

$$S^\oplus = \text{P\_C\_ProductSmash}(\llbracket S_k^\oplus \rrbracket) \quad S^\ominus = \text{P\_C\_ProductSmash}(\llbracket S_k^\ominus \rrbracket) \quad (78)$$

given by

$$\text{su}^\oplus : [o_1 \mid \cdots \mid o_n] \mapsto \text{U1\_C\_CodSum}(\llbracket \text{su}_k^\oplus(o_k) \rrbracket) \quad (79)$$

$$\text{su}^\ominus : [p_1 \mid \cdots \mid p_n] \mapsto \text{U1\_C\_CodSum}(\llbracket \text{su}_k^\ominus(p_k) \rrbracket) \quad (80)$$

This construction is described by the schema [SU1\\_C\\_CodSum](#) (Section 26.9.5).

**Definition 22.17**

Given a list of  $n$  morphisms

$$\llbracket \text{sl}_k : \{S_k\} \text{ kdom} \rightarrow_{\text{SPosL}} \text{kcod}_k \rrbracket \quad (81)$$

the parallel composition is

$$\text{SL1\_C\_CodSumSmash}(\llbracket \text{sl}_k \rrbracket) : \{S\} \text{ kdom} \rightarrow_{\text{SPosL}} \text{P\_C\_Sum}(\llbracket \text{kcod}_k \rrbracket) \quad (82)$$

with

$$S^\oplus = \text{P\_C\_ProductSmash}(\llbracket S_k^\oplus \rrbracket) \quad S^\ominus = \text{P\_C\_ProductSmash}(\llbracket S_k^\ominus \rrbracket) \quad (83)$$

given by

$$\text{sl}^\oplus : [o_1 \mid \cdots \mid o_n] \mapsto \text{L1\_C\_CodSumSmash}(\llbracket \text{sl}_k^\oplus(o_k) \rrbracket) \quad (84)$$

$$\text{sl}^\ominus : [p_1 \mid \cdots \mid p_n] \mapsto \text{L1\_C\_CodSumSmash}(\llbracket \text{sl}_k^\ominus(p_k) \rrbracket) \quad (85)$$

This construction is described by the schema [SL1\\_C\\_CodSumSmash](#) (Section 26.8.6).

**Definition 22.18**

Given a list of  $n$  morphisms

$$\llbracket \text{su}_k : \{S_k\} \text{ kdom} \rightarrow_{\text{SPosU}} \text{kcod}_k \rrbracket \quad (86)$$

the parallel composition is

$$\text{SU1\_C\_CodSumSmash}(\llbracket \text{su}_k \rrbracket) : \{S\} \text{ kdom} \rightarrow_{\text{SPosU}} \text{P\_C\_Sum}(\llbracket \text{kcod}_k \rrbracket) \quad (87)$$

with

$$S^\oplus = \text{P\_C\_ProductSmash}(\llbracket S_k^\oplus \rrbracket) \quad S^\ominus = \text{P\_C\_ProductSmash}(\llbracket S_k^\ominus \rrbracket) \quad (88)$$

given by

$$\text{su}^\oplus : [o_1 \mid \cdots \mid o_n] \mapsto \text{U1\_C\_CodSumSmash}(\llbracket \text{su}_k^\oplus(o_k) \rrbracket) \quad (89)$$

$$\text{su}^\ominus : [p_1 \mid \cdots \mid p_n] \mapsto \text{U1\_C\_CodSumSmash}(\llbracket \text{su}_k^\ominus(p_k) \rrbracket) \quad (90)$$

This construction is described by the schema **SU1\_C\_CodSumSmash** (Section 26.9.6).

## 22.10. Product intersection

### Definition 22.19

Given a list of  $n$  morphisms

$$[sl_k : \{S_k\} \text{ kdom} \rightarrow_{\text{SPosL}} \text{ kcod}] \quad (91)$$

the parallel composition is

$$\text{SL1\_C\_ProdIntersection}([sl_k]) : \{S\} \text{ kdom} \rightarrow_{\text{SPosL}} \text{ kcod} \quad (92)$$

with

$$S^{\ominus} = \text{P\_C\_ProductSmash}([S^{\ominus}_k]) \quad S^{\ominus} = \text{P\_C\_ProductSmash}([S^{\ominus}_k]) \quad (93)$$

given by

$$sl^{\ominus} : [o_1 \mid \dots \mid o_n] \mapsto \text{L1\_C\_Intersection}([sl_k^{\ominus}(o_k)]) \quad (94)$$

$$sl^{\ominus} : [p_1 \mid \dots \mid p_n] \mapsto \text{L1\_C\_Intersection}([sl_k^{\ominus}(p_k)]) \quad (95)$$

This construction is described by the schema **SL1\_C\_ProdIntersection** (Section 26.8.7).

### Definition 22.20

Given a list of  $n$  morphisms

$$[su_k : \{S_k\} \text{ kdom} \rightarrow_{\text{SPosU}} \text{ kcod}_k] \quad (96)$$

the parallel composition is

$$\text{SU1\_C\_ProdIntersection}([su_k]) : \{S\} \text{ kdom} \rightarrow_{\text{SPosU}} \text{ kcod} \quad (97)$$

with

$$S^{\ominus} = \text{P\_C\_ProductSmash}([S^{\ominus}_k]) \quad S^{\ominus} = \text{P\_C\_ProductSmash}([S^{\ominus}_k]) \quad (98)$$

given by

$$su^{\ominus} : [o_1 \mid \dots \mid o_n] \mapsto \text{U1\_C\_Intersection}([su_k^{\ominus}(o_k)]) \quad (99)$$

$$su^{\ominus} : [p_1 \mid \dots \mid p_n] \mapsto \text{U1\_C\_Intersection}([su_k^{\ominus}(p_k)]) \quad (100)$$

This construction is described by the schema **SU1\_C\_ProdIntersection** (Section 26.9.7).

## 22.11. Scalable inverse of sum and multiplication operations

These maps are wrappers around the maps defined in Section 20.6.

### Definition 22.21

The map

$$\text{SU1\_InvSum}(\mathbf{P}, n) : \{\mathbb{N}\} \mathbf{P} \rightarrow_{\text{SPosU}} \mathbf{P}^n \quad (101)$$

is given by the two maps

$$su^{\ominus} : r \mapsto \text{U1\_InvSum\_Opt}(\mathbf{P}, n, r) \quad (102)$$

$$su^{\ominus} : r \mapsto \text{U1\_InvSum\_Pes}(\mathbf{P}, n, r) \quad (103)$$

This construction is described by the schema **SU1\_InvSum** (Section 26.9.16).

**Definition 22.22**

The map

$$SL1\_InvSum(P, n) : \{\mathbb{N}\} P \rightarrow_{\mathbf{SPosL}} P^n \quad (104)$$

is given by the two maps

$$sl^{\ominus} : r \mapsto SL1\_InvSum\_Opt(P, n, r) \quad (105)$$

$$sl^{\oplus} : r \mapsto SL1\_InvSum\_Pes(P, n, r) \quad (106)$$

*This construction is described by the schema  $SL1\_InvSum$  (Section 26.8.16).*

**Lemma 22.23.** We have constructed these maps so that they can be used to approximate in a resolution-complete way the queries induced by the lifted versions of [add](#).

$$SU1\_InvSum(P, n) \in FR_f^{\star}(DP\_LiftL\ add^n) \quad (107)$$

$$SL1\_InvSum(P, n) \in RF_f^{\star}(DP\_LiftU\ add^n) \quad (108)$$

**Definition 22.24**

The map

$$SU1\_InvMultiply(P, n) : \{\mathbb{N}\} P \rightarrow_{\mathbf{SPosU}} P^n \quad (109)$$

is given by the two maps

$$su^{\ominus} : r \mapsto U1\_InvMul\_Opt(P, n, r) \quad (110)$$

$$su^{\oplus} : r \mapsto U1\_InvMul\_Pes(P, n, r) \quad (111)$$

*This construction is described by the schema  $SU1\_InvMultiply$  (Section 26.9.15).*

**Definition 22.25**

The map

$$SL1\_InvMultiply(P, n) : \{\mathbb{N}\} P \rightarrow_{\mathbf{SPosL}} P^n \quad (112)$$

is given by the two maps

$$sl^{\ominus} : r \mapsto L1\_InvMul\_Opt(P, n, r) \quad (113)$$

$$sl^{\oplus} : r \mapsto L1\_InvMul\_Pes(P, n, r) \quad (114)$$

*This construction is described by the schema  $SL1\_InvMultiply$  (Section 26.8.15).*

**Lemma 22.26.**

$$SU1\_InvMultiply(P, n) \in FR_f^{\star}(DP\_LiftL\ mul^n) \quad (115)$$

$$SL1\_InvMultiply(P, n) \in RF_f^{\star}(DP\_LiftU\ mul^n) \quad (116)$$

## 22.12. Explicit approximation

**Definition 22.27** (Constructing multi-resolution  $\mathbf{SPosL}$ )

Fixed a poset  $kdom$  and a poset  $kcod$ , and given

- A chain of  $m$  morphisms  $\llbracket sl_k^{\oplus} \rrbracket$  in  $\mathbf{PosLI}(kdom, kcod)$
- A chain of  $n$  morphisms  $\llbracket sl_k^{\ominus} \rrbracket$  in  $\mathbf{PosLI}(kdom, kcod)^{op}$

we define the morphism

$$SL1\_C\_ExplicitApprox(\llbracket sl_k^{\oplus} \rrbracket, \llbracket sl_k^{\ominus} \rrbracket) : kdom \rightarrow_{\mathbf{SPosLI}} kcod \quad (117)$$

by

$$S^{\ominus} = \llbracket 1, 2, \dots, m \rrbracket \quad (118)$$

$$S^{\oplus} = \llbracket 1, 2, \dots, n \rrbracket \quad (119)$$



$$sl^{\oplus} : i \mapsto sl_i^{\oplus} \quad (120)$$

$$sl^{\oplus} : j \mapsto sl_j^{\oplus} \quad (121)$$

This construction is described by the schema `SL1_C_ExplicitApprox` (Section 26.8.17).

**Definition 22.28** (Constructing multi-resolution **SPosU**)

Fixed a poset `kdom` and a poset `kcod`, and given

- A chain of  $m$  morphisms  $\llbracket su_k^{\oplus} \rrbracket$  in  $\mathbf{PosUI}(kdom, kcod)$
- A chain of  $n$  morphisms  $\llbracket su_k^{\oplus} \rrbracket$  in  $\mathbf{PosUI}(kdom, kcod)^{op}$

we define the morphism

$$SU1\_C\_ExplicitApprox(\llbracket su_k^{\oplus} \rrbracket, \llbracket su_k^{\oplus} \rrbracket) : kdom \rightarrow_{\mathbf{SPosUI}} kcod \quad (122)$$

by

$$S^{\oplus} = \llbracket 1, 2, \dots, m \rrbracket \quad (123)$$

$$S^{\oplus} = \llbracket 1, 2, \dots, n \rrbracket \quad (124)$$

$$su^{\oplus} : i \mapsto su_i^{\oplus} \quad (125)$$

$$su^{\oplus} : j \mapsto su_j^{\oplus} \quad (126)$$

This construction is described by the schema `SU1_C_ExplicitApprox` (Section 26.9.17).

## 23. SPosLI and SPosUI catalog

### 23.1. Lifts

**Definition 23.1** (Lift of a PosLI to a SPosLI)

Given a PosLI morphism

$$\ell : \text{ldom} \rightarrow_{\text{PosLI}} \text{kcod} \{\text{kimp}\} \quad (1)$$

we can lift it as a morphism of SPosLI

$$\text{SL\_L\_Exact}(\ell) : \{1\} \text{ldom} \rightarrow_{\text{SPosLI}} \text{kcod} \{\text{kimp}\} \quad (2)$$

by setting

$$\text{sl}^{\ominus} : * \mapsto \ell \quad (3)$$

$$\text{sl}^{\ominus} : * \mapsto \ell \quad (4)$$

This construction is described by the schema SL\_L\_Exact (Section 26.10.11).

**Definition 23.2** (Lift of a PosUI to a SPosUI)

Given a PosUI morphism

$$u : \text{ldom} \rightarrow_{\text{PosUI}} \text{kcod}, \quad (5)$$

we can lift it as a morphism of SPosUI

$$\text{SU\_L\_Exact}(u) : \{1\} \text{ldom} \rightarrow_{\text{SPosUI}} \text{kcod} \{\text{kimp}\} \quad (6)$$

by setting

$$\text{su}^{\ominus} : * \mapsto u \quad (7)$$

$$\text{su}^{\ominus} : * \mapsto u \quad (8)$$

This construction is described by the schema SU\_L\_Exact (Section 26.11.11).

### 23.2. Explicit approximations

**Definition 23.3** (Constructing multi-resolution SPosLI)

Fixed a poset kdom and a poset kcod, and given

- A chain of  $m$  morphisms  $\llbracket \text{sl}_k^{\ominus} \rrbracket$  in PosLI(kdom, kcod)
- A chain of  $n$  morphisms  $\llbracket \text{sl}_k^{\ominus} \rrbracket$  in PosLI(kdom, kcod)<sup>op</sup>

we define the morphism

$$\text{SL\_L\_Explicit\_Approx}(\llbracket \text{sl}_k^{\ominus} \rrbracket, \llbracket \text{sl}_k^{\ominus} \rrbracket) : \text{ldom} \rightarrow_{\text{SPosLI}} \text{kcod} \quad (9)$$

by

$$\text{S}^{\ominus} = \llbracket 1, 2, \dots, m \rrbracket \quad (10)$$

$$\text{S}^{\ominus} = \llbracket 1, 2, \dots, n \rrbracket \quad (11)$$

$$\text{sl}^{\ominus} : i \mapsto \text{sl}_i^{\ominus} \quad (12)$$

$$\text{sl}^{\ominus} : j \mapsto \text{sl}_j^{\ominus} \quad (13)$$

This construction is described by the schema SL\_L\_Explicit\_Approx (Section 26.10.12).

**Definition 23.4** (Constructing multi-resolution **SPosUI**)

Fixed a poset **kdom** and a poset **kcod**, and given

- A chain of  $m$  morphisms  $\llbracket \text{su}_k^\ominus \rrbracket$  in **PosUI**(**kdom**, **kcod**)
- A chain of  $n$  morphisms  $\llbracket \text{su}_k^\oplus \rrbracket$  in **PosUI**(**kdom**, **kcod**)<sup>op</sup>

we define the morphism

$$\text{SU\_L\_Explicit\_Approx}(\llbracket \text{su}_k^\ominus \rrbracket, \llbracket \text{su}_k^\oplus \rrbracket) : \text{kdom} \rightarrow_{\text{SPosUI}} \text{kcod} \quad (14)$$

by

$$S^\ominus = \llbracket 1, 2, \dots, m \rrbracket \quad (15)$$

$$S^\oplus = \llbracket 1, 2, \dots, n \rrbracket \quad (16)$$

$$\text{su}^\ominus : i \mapsto \text{su}_i^\ominus \quad (17)$$

$$\text{su}^\oplus : j \mapsto \text{su}_j^\oplus \quad (18)$$

This construction is described by the schema **SU\_L\_Explicit\_Approx** (Section 26.11.12).

### 23.3. Parallel composition

**Definition 23.5**

Given a list of  $n$  morphisms

$$\llbracket \text{sl}_k : \{S_k\} \text{kdom}_k \rightarrow_{\text{SPosLI}} \text{kcod}_k \{\text{kimp}_k\} \rrbracket \quad (19)$$

the parallel composition is

$$\text{SL\_C\_Parallel}(\llbracket \text{sl}_k \rrbracket) : \{S\} \text{P\_C\_Product}(\llbracket \text{kdom}_k \rrbracket) \rightarrow_{\text{SPosLI}} \text{P\_C\_Product}(\llbracket \text{kcod}_k \rrbracket) \{\text{P\_C\_ProductSmash}(\llbracket \text{kimp}_k \rrbracket)\} \quad (20)$$

with

$$S^\ominus = \text{P\_C\_ProductSmash}(\llbracket S^\ominus_k \rrbracket) \quad S^\oplus = \text{P\_C\_ProductSmash}(\llbracket S^\oplus_k \rrbracket) \quad (21)$$

given by

$$\text{sl}^\ominus : \langle o_1, \dots, o_n \rangle \mapsto \text{L\_C\_Parallel}(\llbracket \text{sl}_k^\ominus(o_k) \rrbracket) \quad (22)$$

$$\text{sl}^\oplus : \langle p_1, \dots, p_n \rangle \mapsto \text{L\_C\_Parallel}(\llbracket \text{sl}_k^\oplus(p_k) \rrbracket) \quad (23)$$

This construction is described by the schema **SL\_C\_Parallel** (Section 26.10.4).

**Definition 23.6**

Given a list of  $n$  morphisms

$$\llbracket \text{su}_k : \{S_k\} \text{kdom}_k \rightarrow_{\text{SPosUI}} \text{kcod}_k \{\text{kimp}_k\} \rrbracket \quad (24)$$

the parallel composition is

$$\text{SU\_C\_Parallel}(\llbracket \text{su}_k \rrbracket) : \{S\} \text{P\_C\_Product}(\llbracket \text{kdom}_k \rrbracket) \rightarrow_{\text{SPosUI}} \text{P\_C\_Product}(\llbracket \text{kcod}_k \rrbracket) \{\text{P\_C\_ProductSmash}(\llbracket \text{kimp}_k \rrbracket)\} \quad (25)$$

with

$$S^\ominus = \text{P\_C\_ProductSmash}(\llbracket S^\ominus_k \rrbracket) \quad S^\oplus = \text{P\_C\_ProductSmash}(\llbracket S^\oplus_k \rrbracket) \quad (26)$$

given by

$$\text{su}^\ominus : [o_1 \mid \dots \mid o_n] \mapsto \text{U\_C\_Parallel}(\llbracket \text{su}_k^\ominus(o_k) \rrbracket) \quad (27)$$

$$\text{su}^\oplus : [p_1 \mid \dots \mid p_n] \mapsto \text{U\_C\_Parallel}(\llbracket \text{su}_k^\oplus(p_k) \rrbracket) \quad (28)$$

This construction is described by the schema **SU\_C\_Parallel** (Section 26.11.4).

## 23.4. Series composition

### Definition 23.7

Given a list of  $n$  morphisms

$$\llbracket \text{sl}_k : \{S_k\} \text{ kdom}_k \rightarrow_{\text{SPosLI}} \text{ kcod}_k \{ \text{kimp}_k \} \rrbracket \quad (29)$$

such that for all  $k = 1, \dots, n-1$ , we have  $\text{kcod}_k \subseteq \text{kdom}_{k+1}$ , the series composition is

$$\text{SL\_C\_Series}(\llbracket \text{sl}_k \rrbracket) : \{S\} \text{ kdom}_1 \rightarrow_{\text{SPosLI}} \text{ kcod}_n \{ \text{P\_C\_ProductSmash}(\llbracket \text{kimp}_k \rrbracket) \} \quad (30)$$

with

$$S^\ominus = \text{P\_C\_ProductSmash}(\llbracket S^\ominus_k \rrbracket) \quad S^\ominus = \text{P\_C\_ProductSmash}(\llbracket S^\ominus_k \rrbracket) \quad (31)$$

given by

$$\text{sl}^\ominus : [o_1 \mid \dots \mid o_n] \mapsto \text{L\_C\_Series}(\llbracket \text{sl}_k^\ominus(o_k) \rrbracket) \quad (32)$$

$$\text{sl}^\ominus : [p_1 \mid \dots \mid p_n] \mapsto \text{L\_C\_Series}(\llbracket \text{sl}_k^\ominus(p_k) \rrbracket) \quad (33)$$

This construction is described by the schema **SL\_C\_Series** (Section 26.10.5).

### Definition 23.8

Given a list of  $n$  morphisms

$$\llbracket \text{su}_k : \{S_k\} \text{ kdom}_k \rightarrow_{\text{SPosUI}} \text{ kcod}_k \{ \text{kimp}_k \} \rrbracket \quad (34)$$

the series composition is

$$\text{SU\_C\_Series}(\llbracket \text{su}_k \rrbracket) : \{S\} \text{ kdom} \rightarrow_{\text{SPosUI}} \text{ kcod} \{ \text{P\_C\_ProductSmash}(\llbracket \text{kimp}_k \rrbracket) \} \quad (35)$$

with

$$S^\ominus = \text{P\_C\_ProductSmash}(\llbracket S^\ominus_k \rrbracket) \quad S^\ominus = \text{P\_C\_ProductSmash}(\llbracket S^\ominus_k \rrbracket) \quad (36)$$

given by

$$\text{su}^\ominus : [o_1 \mid \dots \mid o_n] \mapsto \text{U\_C\_Series}(\llbracket \text{su}_k^\ominus(o_k) \rrbracket) \quad (37)$$

$$\text{su}^\ominus : [p_1 \mid \dots \mid p_n] \mapsto \text{U\_C\_Series}(\llbracket \text{su}_k^\ominus(p_k) \rrbracket) \quad (38)$$

This construction is described by the schema **SU\_C\_Series** (Section 26.11.5).

## 23.5. Intersection

### Definition 23.9

Given two posets  $\text{kdom}$  and  $\text{kcod}$  and a list of  $n$  morphisms

$$\llbracket \text{sl}_k : \{S_k\} \text{ kdom} \rightarrow_{\text{SPosLI}} \text{ kcod} \{ \text{kimp}_k \} \rrbracket \quad (39)$$

the intersection composition is

$$\text{SL\_C\_Intersection}(\llbracket \text{sl}_k \rrbracket) : \{S\} \text{ kdom} \rightarrow_{\text{SPosLI}} \text{ kcod} \{ \text{P\_C\_ProductSmash}(\llbracket \text{kimp}_k \rrbracket) \} \quad (40)$$

with

$$S^\ominus = \text{P\_C\_ProductSmash}(\llbracket S^\ominus_k \rrbracket) \quad S^\ominus = \text{P\_C\_ProductSmash}(\llbracket S^\ominus_k \rrbracket) \quad (41)$$

given by

$$\text{sl}^\ominus : [o_1 \mid \dots \mid o_n] \mapsto \text{L\_C\_Intersection}(\llbracket \text{sl}_k^\ominus(o_k) \rrbracket) \quad (42)$$

$$\text{sl}^\ominus : [p_1 \mid \dots \mid p_n] \mapsto \text{L\_C\_Intersection}(\llbracket \text{sl}_k^\ominus(p_k) \rrbracket) \quad (43)$$

This construction is described by the schema **SL\_C\_Intersection** (Section 26.10.3).

**Definition 23.10**

Given two posets  $\mathbf{kdom}$  and  $\mathbf{kcod}$  and a list of  $n$  morphisms

$$\llbracket \mathbf{su}_k : \{S_k\} \mathbf{kdom} \rightarrow_{\mathbf{sPosUI}} \mathbf{kcod} \{\mathbf{kimp}_k\} \rrbracket \quad (44)$$

the intersection composition is

$$\mathbf{SU\_C\_Intersection}(\llbracket \mathbf{su}_k \rrbracket) : \{S\} \mathbf{kdom} \rightarrow_{\mathbf{sPosUI}} \mathbf{kcod} \{\mathbf{P\_C\_ProductSmash}(\llbracket \mathbf{kimp}_k \rrbracket)\} \quad (45)$$

with

$$S^\ominus = \mathbf{P\_C\_ProductSmash}(\llbracket S_k^\ominus \rrbracket) \quad S^\ominus = \mathbf{P\_C\_ProductSmash}(\llbracket S_k^\ominus \rrbracket) \quad (46)$$

given by

$$\mathbf{su}^\ominus : [o_1 \mid \dots \mid o_n] \mapsto \mathbf{U\_C\_Intersection}(\llbracket \mathbf{su}_k^\ominus(o_k) \rrbracket) \quad (47)$$

$$\mathbf{su}^\ominus : [p_1 \mid \dots \mid p_n] \mapsto \mathbf{U\_C\_Intersection}(\llbracket \mathbf{su}_k^\ominus(p_k) \rrbracket) \quad (48)$$

*This construction is described by the schema  $\mathbf{SU\_C\_Intersection}$  (Section 26.11.3).*

## 23.6. Union

**Definition 23.11**

Given two posets  $\mathbf{kdom}$  and  $\mathbf{kcod}$  and a list of maps

$$\llbracket \mathbf{sl}_k : \{S_k\} \mathbf{kdom} \rightarrow_{\mathbf{sPosLI}} \mathbf{kcod} \{\mathbf{kimp}_k\} \rrbracket \quad (49)$$

the union composition is

$$\mathbf{SL\_C\_Union}(\llbracket \mathbf{sl}_k \rrbracket) : \{S\} \mathbf{kdom} \rightarrow_{\mathbf{sPosLI}} \mathbf{kcod} \{\mathbf{P\_C\_ProductSmash}(\llbracket \mathbf{kimp}_k \rrbracket)\} \quad (50)$$

with

$$S^\ominus = \mathbf{P\_C\_ProductSmash}(\llbracket S_k^\ominus \rrbracket) \quad S^\ominus = \mathbf{P\_C\_ProductSmash}(\llbracket S_k^\ominus \rrbracket) \quad (51)$$

given by

$$\mathbf{sl}^\ominus : [o_1 \mid \dots \mid o_n] \mapsto \mathbf{L\_C\_Union}(\llbracket \mathbf{sl}_k^\ominus(o_k) \rrbracket) \quad (52)$$

$$\mathbf{sl}^\ominus : [p_1 \mid \dots \mid p_n] \mapsto \mathbf{L\_C\_Union}(\llbracket \mathbf{sl}_k^\ominus(p_k) \rrbracket) \quad (53)$$

*This construction is described by the schema  $\mathbf{SL\_C\_Union}$  (Section 26.10.6).*

**Definition 23.12**

Given two posets  $\mathbf{kdom}$  and  $\mathbf{kcod}$  and a list of maps

$$\llbracket \mathbf{su}_k : \{S_k\} \mathbf{kdom} \rightarrow_{\mathbf{sPosUI}} \mathbf{kcod} \{\mathbf{kimp}_k\} \rrbracket \quad (54)$$

the union composition is the morphism

$$\mathbf{SU\_C\_Union}(\llbracket \mathbf{su}_k \rrbracket) : \{S\} \mathbf{kdom} \rightarrow_{\mathbf{sPosUI}} \mathbf{kcod} \{\mathbf{P\_C\_ProductSmash}(\llbracket \mathbf{kimp}_k \rrbracket)\} \quad (55)$$

with

$$S^\ominus = \mathbf{P\_C\_ProductSmash}(\llbracket S_k^\ominus \rrbracket) \quad S^\ominus = \mathbf{P\_C\_ProductSmash}(\llbracket S_k^\ominus \rrbracket) \quad (56)$$

given by

$$\mathbf{su}^\ominus : [o_1 \mid \dots \mid o_n] \mapsto \mathbf{U\_C\_Union}(\llbracket \mathbf{su}_k^\ominus(o_k) \rrbracket) \quad (57)$$

$$\mathbf{su}^\ominus : [p_1 \mid \dots \mid p_n] \mapsto \mathbf{U\_C\_Union}(\llbracket \mathbf{su}_k^\ominus(p_k) \rrbracket) \quad (58)$$

*This construction is described by the schema  $\mathbf{SU\_C\_Union}$  (Section 26.11.6).*

## 23.7. Trace

### Definition 23.13

Given a morphism

$$sl_0 : \{S\} P\_C\_Product(\llbracket kdom_1, kdom_2 \rrbracket) \rightarrow_{SPosLI} P\_C\_Product(\llbracket kcod_1, kcod_2 \rrbracket) \{kimp\} \quad (59)$$

such that  $kcod_2 \subseteq kdom_2$  we define the morphism

$$SL\_C\_Trace(sl) : \{S\} kdom_1 \rightarrow_{SPosLI} kcod_1 \{kimp\} \quad (60)$$

given by

$$sl^{\oplus} : o \mapsto L\_C\_Trace(sl_0^{\oplus}(o)) \quad (61)$$

$$sl^{\oplus} : p \mapsto L\_C\_Trace(sl_0^{\oplus}(p)) \quad (62)$$

*This construction is described by the schema  $SL\_C\_Trace$  (Section 26.10.9).*

### Definition 23.14

Given a morphism

$$su_0 : \{S\} P\_C\_Product(\llbracket kdom_1, kdom_2 \rrbracket) \rightarrow_{SPosUI} P\_C\_Product(\llbracket kcod_1, kcod_2 \rrbracket) \{kimp\} \quad (63)$$

such that  $kcod_2 \subseteq kdom_2$  we define the morphism

$$SU\_C\_Trace(su) : \{S\} kdom_1 \rightarrow_{SPosUI} kcod_1 \{kimp\} \quad (64)$$

given by

$$su^{\oplus} : o \mapsto U\_C\_Trace(su_0^{\oplus}(o)) \quad (65)$$

$$su^{\oplus} : p \mapsto U\_C\_Trace(su_0^{\oplus}(p)) \quad (66)$$

*This construction is described by the schema  $SU\_C\_Trace$  (Section 26.11.9).*

## 24. DP catalog

### 24.1. Identity

**Definition 24.1** (Identity)

Given a poset  $\mathbf{P}$ , the identity DP is defined as

$$\begin{aligned} \text{DP\_Identity}(\mathbf{P}) : \mathbf{P} &\rightarrow_{\text{DP}} \mathbf{P} \\ \langle f^*, r \rangle &\mapsto f^* \leq_{\mathbf{P}} r \end{aligned} \quad (1)$$

This construction is described by the schema  $\text{DP\_Identity}$  (Section 26.12.2).

**Lemma 24.2** (Query solutions for the identity).

$$\text{FR DP\_Identity}(\mathbf{P}) = \text{U1\_Identity}(\mathbf{P}) \quad (2)$$

$$\text{RF DP\_Identity}(\mathbf{P}) = \text{L1\_Identity}(\mathbf{P}) \quad (3)$$

$$(4)$$

This construction is a particular case of  $\text{DP\_LiftL}$  and  $\text{DP\_Iso}$ :

$$\text{DP\_Identity}(\mathbf{P}) = \text{DP\_LiftL}(\text{id}_{\mathbf{P}}) \quad (5)$$

$$= \text{DP\_Iso}(\text{id}_{\mathbf{P}}) \quad (6)$$

### 24.2. Ambient conversion

**Definition 24.3** (Ambient conversion)

Given a poset  $\mathbf{P}$  and two posets  $\mathbf{F}, \mathbf{R} \subseteq \mathbf{P}$ , the ambient conversion DP is defined as

$$\begin{aligned} \text{DP\_AmbientConversion}(\mathbf{P}, \mathbf{F}, \mathbf{R}) : \mathbf{F} &\rightarrow_{\text{DP}} \mathbf{R} \\ \langle f^*, r \rangle &\mapsto f^* \leq_{\mathbf{P}} r \end{aligned} \quad (7)$$

**Lemma 24.4** (Query solutions of  $\text{DP\_AmbientConversion}$ ).

$$\text{FR DP\_AmbientConversion}(\mathbf{P}, \mathbf{F}, \mathbf{R}) = \text{U1\_RepresentPrincipalUpperSet}(\mathbf{P}, \mathbf{F}, \mathbf{R}) \quad (8)$$

$$\text{RF DP\_AmbientConversion}(\mathbf{P}, \mathbf{F}, \mathbf{R}) = \text{L1\_RepresentPrincipalLowerSet}(\mathbf{P}, \mathbf{R}, \mathbf{F}) \quad (9)$$

This construction is described by the schema  $\text{DP\_AmbientConversion}$  (Section 26.12.5).

### 24.3. Isomorphism

**Definition 24.5** (Isomorphism of posets)

Given two posets  $\mathbf{F}$  and  $\mathbf{R}$ , and an order isomorphism

$$\langle g, g^{-1} \rangle : \mathbf{F} \leftrightarrow_{\text{Pos}} \mathbf{R}, \quad (10)$$

we can define the DP

$$\begin{aligned} \text{DP\_Iso}(\mathbf{F}, \mathbf{R}) : \mathbf{P} &\rightarrow_{\text{DP}} \mathbf{P} \\ \langle f^*, r \rangle &\mapsto f^* \leq_{\mathbf{P}} g^{-1}(r) = g(f^*) \leq_{\mathbf{P}} r \end{aligned} \quad (11)$$

This construction is described by the schema **DP\_Iso** (Section 26.12.7).

**Lemma 24.6** (Isomorphisms of **DP\_Iso**).

$$\text{DP\_Iso}(g) \circ \text{DP\_Iso}(h) \cong_{\mathbf{B}} \text{DP\_Iso}(g \circ h) \quad (12)$$

**Lemma 24.7** (Query solutions for **DP\_Iso**).

$$\text{FR}(\text{DP\_Iso}(g)) = \text{U1\_Lift}(g) \quad (13)$$

$$\text{RF}(\text{DP\_Iso}(g)) = \text{L1\_Lift}(g^{-1}) \quad (14)$$

$$(15)$$

## 24.4. Lower lift of a map

**Definition 24.8** (Lower lift of a monotone map to a **DP**)

Given a monotone map

$$h : \mathbf{R} \rightarrow_{\text{Pos}} \mathbf{F}, \quad (16)$$

we can lift it to a **DP**

$$\begin{aligned} \text{DP\_LiftL}(h) : \mathbf{F} &\rightarrow_{\mathbf{DP}} \mathbf{R} \\ \langle f^*, r \rangle &\mapsto f^* \leq_{\mathbf{F}} h(r) \end{aligned} \quad (17)$$

**Lemma 24.9** (Query solutions for **DP\_LiftL**).

$$\text{FR}(\text{DP\_LiftL}(h)) = \text{Ui } h \quad (18)$$

$$\text{RF}(\text{DP\_LiftL}(h)) = \text{L1\_Lift}(h) \quad (19)$$

$$(20)$$

**Lemma 24.10** (Isomorphisms of **DP\_LiftL**).

$$\text{DP\_LiftL}(g) \circ \text{DP\_LiftL}(h) \cong_{\mathbf{B}} \text{DP\_LiftL}(h \circ g) \quad (21)$$

Note that the order of the arguments in the composition is reversed.

## 24.5. Upper lift of a map

**Definition 24.11** (Upper lift of a monotone map to a **DP**)

Given a monotone map

$$h : \mathbf{F}^{\text{op}} \rightarrow_{\text{Pos}} \mathbf{R}, \quad (22)$$

we can lift it to a **DP**

$$\begin{aligned} \text{DP\_LiftU}(h) : \mathbf{F} &\rightarrow_{\mathbf{DP}} \mathbf{R} \\ \langle f^*, r \rangle &\mapsto h(f^*) \leq_{\mathbf{R}} r \end{aligned} \quad (23)$$

This construction is described by the schema **DP\_LiftU** (Section 26.12.9).

**Lemma 24.12** (Isomorphisms of **DP\_LiftU**).

$$\text{DP\_LiftU}(g) \circ \text{DP\_LiftU}(h) \cong_{\mathbf{B}} \text{DP\_LiftU}(g \circ h) \quad (24)$$

**Lemma 24.13** (Query solutions for **DP\_LiftU**).

$$\text{FR}(\text{DP\_LiftU}(g)) = \text{U1\_Lift}(g) \quad (25)$$

$$\text{RF}(\text{DP\_LiftU}(g)) = \text{Li } g \quad (26)$$



## 24.6. Functionalities/requirements limits

In this section we specify several DP constructions that are used as “plumbing” when compiling a MCDP.

This is not a minimal set: to have an efficient compiler, it is actually necessary to recognize useful special cases for which queries are simpler to solve.

Most of these also only make sense when the posets in question are not lattices. If the posets are lattices then these constructions simplify. We show some of the simplifications below.

### 24.6.1. Functionality not more than the requirement and constant

**Definition 24.14** (Functionality not more than the requirement and constant)  
Given two posets  $\mathbf{F}, \mathbf{R} \subseteq \mathbf{P}$ , and a constant  $c \in \mathbf{P}$ , we define the DP

$$\begin{aligned} \text{DP\_FuncNotMoreThan}(\mathbf{P}, \mathbf{F}, \mathbf{R}, c) : \mathbf{F} &\rightarrow_{\text{DP}} \mathbf{R} \\ \langle f^*, r \rangle &\mapsto (f^* \leq_{\mathbf{P}} r) \wedge (f^* \leq_{\mathbf{P}} c) \end{aligned} \quad (27)$$

*This construction is described by the schema  $\text{DP\_FuncNotMoreThan}$  (Section 26.12.15).*

**Lemma 24.15.**

$$\text{RF DP\_FuncNotMoreThan}(\mathbf{P}, \mathbf{F}, \mathbf{R}, c) = \text{M\_Threshold1}(\mathbf{P}, c) \quad (28)$$

### 24.6.2. Requirement not less than the functionality and constant

**Definition 24.16** (Requirement not less than the functionality and constant)  
Given two posets  $\mathbf{F}, \mathbf{R} \subseteq \mathbf{P}$ , and a constant  $c \in \mathbf{P}$ , we define the DP

$$\begin{aligned} \text{DP\_ResNotLessThan}(\mathbf{P}, \mathbf{F}, \mathbf{R}, c) : \mathbf{F} &\rightarrow_{\text{DP}} \mathbf{R} \\ \langle f^*, r \rangle &\mapsto (f^* \leq_{\mathbf{P}} r) \wedge (c \leq_{\mathbf{P}} r) \end{aligned} \quad (29)$$

*This construction is described by the schema  $\text{DP\_ResNotLessThan}$  (Section 26.12.16).*

Note that if  $\mathbf{P}$  was a lattice, we would have

$$\text{DP\_ResNotLessThan}(\mathbf{P}, \mathbf{F}, \mathbf{R}, c) = \text{DP\_LiftU M\_JoinConstant}(\mathbf{P}, c) \quad (30)$$

**Lemma 24.17.**

$$\text{FR DP\_ResNotLessThan}(\mathbf{P}, \mathbf{F}, \mathbf{R}, c) = \text{M\_Threshold2}(\mathbf{P}, c) \quad (31)$$

### 24.6.3. All functionalities less than the requirement

**Definition 24.18**  
Given  $n$  posets  $\llbracket \mathbf{F}_k \rrbracket$  and  $\mathbf{R}$  all subposets of a poset  $\mathbf{P}$ , we define the DP

$$\begin{aligned} \text{DP\_AllFiLeqR}(\mathbf{P}, \llbracket \mathbf{F}_k \rrbracket, \mathbf{R}) : \text{P\_C\_Product}(\llbracket \mathbf{F}_k \rrbracket) &\rightarrow_{\text{DP}} \mathbf{R} \\ \langle \langle f_1^*, \dots, f_n^* \rangle, r \rangle &\mapsto \bigwedge_k (f_k^* \leq_{\mathbf{P}} r) \end{aligned} \quad (32)$$

If  $\mathbf{P}$  was a join semilattice, we would have

$$\bigwedge_k (f_k^* \leq_{\mathbf{P}} r) = (\bigvee_k f_k^*) \leq_{\mathbf{P}} r \quad (33)$$

thus

$$\text{DP\_AllFiLeqR}(\mathbf{P}, \llbracket \mathbf{F}_k \rrbracket, \mathbf{R}) = \text{DP\_LiftU M\_Join}(\llbracket \mathbf{F}_k \rrbracket) \quad (34)$$

*This construction is described by the schema  $\text{DP\_All\_Fi\_Leq\_R}$  (Section 26.12.17).*

#### 24.6.4. Any functionality less than the requirement

**Definition 24.19**

Given a poset  $\llbracket \mathbf{F}_k \rrbracket$  and  $\mathbf{R}$  all subposets of a poset  $\mathbf{P}$ , we define the DP

$$\begin{aligned} \text{DP\_Any\_Fi\_Leq\_R}(\mathbf{P}, \llbracket \mathbf{F}_k \rrbracket, \mathbf{R}) : \mathbf{P\_C\_Product}(\llbracket \mathbf{F}_k \rrbracket) &\rightarrow_{\text{DP}} \mathbf{R} \\ \langle \langle f_1^*, \dots, f_n^* \rangle, r \rangle &\mapsto \bigvee_k (f_k^* \leq_{\mathbf{P}} r) \end{aligned} \quad (35)$$

This construction is described by the schema  $\text{DP\_Any\_Fi\_Leq\_R}$  (Section 26.12.18).

#### 24.6.5. All requirements more than the functionality

**Definition 24.20**

Given a list of  $n$  posets  $\llbracket \mathbf{R}_k \rrbracket$  and  $\mathbf{R}$  all subposets of a poset  $\mathbf{P}$ , we define the DP

$$\begin{aligned} \text{DP\_All\_Fi\_Leq\_R}(\mathbf{P}, \mathbf{F}, \llbracket \mathbf{R}_k \rrbracket) : \mathbf{F} &\rightarrow_{\text{DP}} \mathbf{P\_C\_Product}(\llbracket \mathbf{R}_k \rrbracket) \\ \langle f^*, \langle r_1, \dots, r_n \rangle \rangle &\mapsto \bigwedge_k (f^* \leq_{\mathbf{P}} r_k) \end{aligned} \quad (36)$$

This construction is described by the schema  $\text{DP\_F\_Leq\_All\_Ri}$  (Section 26.12.19).

#### 24.6.6. Any requirement more than the functionality

**Definition 24.21**

Given a list of  $n$  posets  $\llbracket \mathbf{R}_k \rrbracket$  and  $\mathbf{F}$  all subposets of a poset  $\mathbf{P}$ , we define the DP

$$\begin{aligned} \text{DP\_F\_Leq\_Any\_Ri}(\mathbf{P}, \mathbf{F}, \llbracket \mathbf{R}_k \rrbracket) : \mathbf{F} &\rightarrow_{\text{DP}} \mathbf{P\_C\_Product}(\llbracket \mathbf{R}_k \rrbracket) \\ \langle f^*, \langle r_1, \dots, r_n \rangle \rangle &\mapsto \bigvee_k (f^* \leq_{\mathbf{P}} r_k) \end{aligned} \quad (37)$$

This construction is described by the schema  $\text{DP\_F\_Leq\_Any\_Ri}$  (Section 26.12.20).

#### 24.6.7. All constants less than the requirement

**Definition 24.22**

Given a poset  $\mathbf{P}$  and a list of  $n$  constants  $\llbracket c_k \rrbracket$  we define the DP

$$\begin{aligned} \text{DP\_All\_Constants\_Leq\_R}(\mathbf{P}, \llbracket c_k \rrbracket) : \mathbb{1} &\rightarrow_{\text{DP}} \mathbf{P} \\ \langle *, r \rangle &\mapsto \bigwedge_k (c_k \leq_{\mathbf{P}} r) \end{aligned} \quad (38)$$

This construction is described by the schema  $\text{DP\_All\_Constants\_Leq\_R}$  (Section 26.12.21).

#### 24.6.8. Functionality less than all constants

**Definition 24.23**

Given a poset  $\mathbf{P}$  and a list of  $n$  constants  $\llbracket c_k \rrbracket$  we define the DP

$$\begin{aligned} \text{DP\_F\_Leq\_All\_Constants}(\mathbf{P}, \llbracket c_k \rrbracket) : \mathbf{P} &\rightarrow_{\text{DP}} \mathbb{1} \\ \langle f^*, * \rangle &\mapsto \bigwedge_k (f^* \leq_{\mathbf{P}} c_k) \end{aligned} \quad (39)$$

This construction is described by the schema  $\text{DP\_F\_Leq\_All\_Constants}$  (Section 26.12.22).

#### 24.6.9. Functionality and all constants less than the requirement

**Definition 24.24**

Given a poset  $\mathbf{P}$  and a list of  $n$  constants  $\llbracket c_k \rrbracket$  we define the DP

$$\text{DP\_All\_Constants\_And\_F\_Leq\_R}(\mathbf{P}, \llbracket c_k \rrbracket) : \mathbf{P} \rightarrow_{\text{DP}} \mathbf{P} \quad (40)$$

$$\langle f^*, r \rangle \mapsto (f^* \leq_{\mathbf{P}} r) \wedge \bigwedge_k (c_k \leq_{\mathbf{P}} r)$$

*This construction is described by the schema  $\text{DP\_All\_Constants\_And\_F\_Leq\_R}$  (Section 26.12.23).*

#### 24.6.10. Functionality or any constant less than the requirement

**Definition 24.25**

Given a poset  $\mathbf{P}$  and a list of  $n$  constants  $\llbracket c_k \rrbracket$  we define the DP

$$\text{DP\_Any\_Constants\_Or\_F\_Leq\_R}(\mathbf{P}, \llbracket c_k \rrbracket) : \mathbf{P} \rightarrow_{\text{DP}} \mathbf{P} \quad (41)$$

$$\langle f^*, r \rangle \mapsto (f^* \leq_{\mathbf{P}} r) \vee \bigvee_k (c_k \leq_{\mathbf{P}} r)$$

*This construction is described by the schema  $\text{DP\_Any\_Constants\_Or\_F\_Leq\_R}$  (Section 26.12.24).*

#### 24.6.11. Functionality less than the requirement and all constants

**Definition 24.26**

Given a poset  $\mathbf{P}$  and a list of  $n$  constants  $\llbracket c_k \rrbracket$  we define the DP

$$\text{DP\_F\_Leq\_All\_R\_And\_Constants}(\mathbf{P}, \llbracket c_k \rrbracket) : \mathbf{P} \rightarrow_{\text{DP}} \mathbf{P} \quad (42)$$

$$\langle f^*, r \rangle \mapsto (f^* \leq_{\mathbf{P}} r) \wedge \bigwedge_k (c_k \leq_{\mathbf{P}} r)$$

*This construction is described by the schema  $\text{DP\_F\_Leq\_All\_R\_And\_Constants}$  (Section 26.12.25).*

#### 24.6.12. Functionality less than the requirement or any constant

**Definition 24.27**

Given a poset  $\mathbf{P}$  and a list of  $n$  constants  $\llbracket c_k \rrbracket$  we define the DP

$$\text{DP\_F\_Leq\_Any\_R\_And\_Constants}(\mathbf{P}, \llbracket c_k \rrbracket) : \mathbf{P} \rightarrow_{\text{DP}} \mathbf{P} \quad (43)$$

$$\langle f^*, r \rangle \mapsto (f^* \leq_{\mathbf{P}} r) \vee \bigvee_k (c_k \leq_{\mathbf{P}} r)$$

*This construction is described by the schema  $\text{DP\_F\_Leq\_Any\_R\_And\_Constants}$  (Section 26.12.26).*

## 25. DPI catalog

### 25.1. True and false

#### 25.1.1. True

The true DPI is the one that allows any combination of functionality and requirements using the same blueprint.

**Definition 25.1**

Given arbitrary posets  $\mathbf{F}, \mathbf{R}, \mathcal{B}$  and a value  $b_0 \in \mathcal{B}$ , we define the DP

$$\text{DP\_True}(\mathbf{F}, \mathbf{R}, \mathcal{B}, b_0) : \mathbf{F} \rightarrow_{\text{DPI}} \mathbf{R}\{\mathcal{B}\} \quad (1)$$

by the data

$$\mathbf{I} = \text{P\_C\_ProductSmash}(\llbracket \mathbf{F}, \mathbf{R}^{\text{op}} \rrbracket) \quad (2)$$

$$\text{prov} : [f \mid r^*] \mapsto f \quad (3)$$

$$\text{req} : [f \mid r^*] \mapsto r^* \quad (4)$$

$$\text{avail} : [f^* \mid r] \mapsto \top \quad (5)$$

$$\text{feas} : [f \mid r^*] \mapsto \top \quad (6)$$

$$\text{IB} : [f \mid r^*] \mapsto b_0 \quad (7)$$

$$(8)$$

*This construction is described by the schema  $\text{DP\_True}$  (Section 26.12.3).*

Note that

$$\text{DP\_True}(\mathbf{F}_1, \mathbf{R}_1, \mathcal{B}_1, b_1) \mathbin{\text{\&}} \text{DP\_True}(\mathbf{F}_2, \mathbf{R}_2, \mathcal{B}_2, b_2) \cong_{\mathbf{B}} \text{DP\_True}(\mathbf{F}_1, \mathbf{R}_2, \text{P\_C\_ProductSmash}(\mathcal{B}_1, \mathcal{B}_2), [b_1 \mid b_2]), \quad (9)$$

however, the expressions  $\text{DP\_True}(\dots) \mathbin{\text{\&}} \mathbf{d}$  and  $\mathbf{d} \mathbin{\text{\&}} \text{DP\_True}(\dots)$  do not simplify.

#### 25.1.2. False

The false DPI is the one that does not allow any combination of functionality and requirements.

**Definition 25.2**

Given arbitrary posets  $\mathbf{F}, \mathbf{R}, \mathcal{B}$  we define the DP

$$\text{DP\_False}(\mathbf{F}, \mathbf{R}, \mathcal{B}) : \mathbf{F} \rightarrow_{\text{DPI}} \mathbf{R}\{\mathcal{B}\} \quad (10)$$

by the data

$$\mathbf{I} = \text{P\_C\_ProductSmash}(\llbracket \mathbf{F}, \mathbf{R}^{\text{op}} \rrbracket) \quad (11)$$

$$\text{prov} : [f \mid r^*] \mapsto f \quad (12)$$

$$\text{req} : [f \mid r^*] \mapsto r^* \quad (13)$$

$$\text{avail} : [f^* \mid r] \mapsto \perp \quad (14)$$

$$\text{feas} : [f \mid r^*] \mapsto \perp \quad (15)$$

$$\text{IB} : \emptyset \rightarrow \emptyset \quad (16)$$

*This construction is described by the schema  $\text{DP\_False}$  (Section 26.12.4).*

Note that we do not need to specify  $\mathbf{IB}$  because the domain is empty.

This construction is useful during compilation, as in certain cases we can prove that the DP is infeasible, in the sense that the compiler can simplify the DP to  $\mathbf{DP\_False}$ .

**Lemma 25.3** (Absorption properties of  $\mathbf{DP\_False}$ ). For any  $\mathbf{d} : \mathbf{F}_2 \rightarrow_{\mathbf{DP}\mathbf{I}} \mathbf{R}_2 \{\mathcal{B}_2\}$

$$\mathbf{DP\_False}((\mathbf{F}_1, \mathbf{R}_1, \mathcal{B}_1) \mathbin{\text{\textcircled{;}}} \mathbf{d} \cong_{\mathbf{B}} \mathbf{DP\_False}((\mathbf{F}_1, \mathbf{R}_2, \mathbf{P\_C\_ProductSmash}(\mathcal{B}_1, \mathcal{B}_2))) \quad (17)$$

and for any  $\mathbf{d} : \mathbf{F}_1 \rightarrow_{\mathbf{DP}\mathbf{I}} \mathbf{R}_1 \{\mathcal{B}_1\}$

$$\mathbf{d} \mathbin{\text{\textcircled{;}}} \mathbf{DP\_False}((\mathbf{F}_2, \mathbf{R}_2, \mathcal{B}_2) \cong_{\mathbf{B}} \mathbf{DP\_False}((\mathbf{F}_1, \mathbf{R}_2, \mathbf{P\_C\_ProductSmash}(\mathcal{B}_1, \mathcal{B}_2))). \quad (18)$$

Similar absorption properties hold for parallel and trace composition.

## 25.2. Catalogs

### Definition 25.4

Given arbitrary posets  $\mathbf{F}, \mathbf{R}, \mathbf{I}_0, \mathcal{B}$  and a list of options

$$\llbracket \langle f_k, r_k^*, i_k, b_k \rangle \rrbracket \subseteq \mathbf{P\_C\_Product}(\llbracket \mathbf{F}, \mathbf{R}^{\text{op}}, \mathbf{I}_0, \mathcal{B} \rrbracket) \quad (19)$$

with the constraint that for all  $a, b, i_a \leq_{\mathbf{I}_0} i_b$  implies  $b_a \leq_{\mathcal{B}} b_b$ , we can define the *catalog*

$$\mathbf{DP\_Catalog} : \mathbf{F} \rightarrow_{\mathbf{DP}\mathbf{I}} \mathbf{R} \{\mathcal{B}\} \quad (20)$$

by choosing the implementation space  $\mathbf{I}$  as the poset that has as carrier the set  $\{\langle i_i, i_i \rangle\}$  and the order given by lexicographic order whose first component is

$$i_i \leq_{\mathbf{I}} i_j \iff (f_i \leq_{\mathbf{F}} f_j) \wedge (r_j \leq_{\mathbf{R}} r_i) \quad (21)$$

and whose second component is the original order on  $\mathbf{I}_0$ . The remaining data is

$$\mathbf{prov} : i_k \mapsto f_k \quad (22)$$

$$\mathbf{req} : i_k \mapsto r_k^* \quad (23)$$

$$\mathbf{avail} : i_k \mapsto \top \quad (24)$$

$$\mathbf{feas} : i_k \mapsto \top \quad (25)$$

$$\mathbf{IB} : i_k \mapsto b_k \quad (26)$$

$$(27)$$

*This construction is described by the schema  $\mathbf{DP\_Catalog}$  (Section 26.12.6).*

The intuition is that the arbitrary poset  $\mathbf{I}_0$  represents an internal “preference” on the implementation space that is independent of the functionality and requirements of each implementation. However, in the DPI we want to first order the implementations by functionality and requirements, and only in case of ties by the internal preference order. With this construction, we are guaranteed that  $\mathbf{prov}, \mathbf{req}, \mathbf{IB}$  are monotone maps.

**Lemma 25.5** (Query solutions for  $\mathbf{DP\_Catalog}$ ).

$$\mathbf{FR}(\mathbf{DP\_Catalog}(\llbracket \langle f_k, r_k^*, i_k, b_k \rangle \rrbracket)) = \mathbf{U1\_Catalog}(\llbracket \langle f_k, r_k^* \rangle \rrbracket) \quad (28)$$

$$\mathbf{FRI}(\mathbf{DP\_Catalog}(\llbracket \langle f_k, r_k^*, i_k, b_k \rangle \rrbracket)) = \mathbf{U\_Catalog}(\llbracket \langle f_k, r_k^*, b_k \rangle \rrbracket) \quad (29)$$

$$\mathbf{FRB}(\mathbf{DP\_Catalog}(\llbracket \langle f_k, r_k^*, i_k, b_k \rangle \rrbracket)) = \mathbf{U\_Catalog}(\llbracket \langle f_k, r_k^*, i_k \rangle \rrbracket) \quad (30)$$

$$\mathbf{RF}(\mathbf{DP\_Catalog}(\llbracket \langle f_k, r_k^*, i_k, b_k \rangle \rrbracket)) = \mathbf{L1\_Catalog}(\llbracket \langle f_k, r_k^* \rangle \rrbracket) \quad (31)$$

$$\mathbf{RFI}(\mathbf{DP\_Catalog}(\llbracket \langle f_k, r_k^*, i_k, b_k \rangle \rrbracket)) = \mathbf{L\_Catalog}(\llbracket \langle f_k, r_k^*, b_k \rangle \rrbracket) \quad (32)$$

$$\mathbf{RFB}(\mathbf{DP\_Catalog}(\llbracket \langle f_k, r_k^*, i_k, b_k \rangle \rrbracket)) = \mathbf{L\_Catalog}(\llbracket \langle f_k, r_k^*, i_k \rangle \rrbracket) \quad (33)$$

The construction  $\mathbf{DP\_GenericConstant}$  is a special case of the construction  $\mathbf{DP\_Catalog}$  which has only a single blueprint  $b_0 \in \mathcal{B}$ .

*This construction is described by the schema  $\mathbf{DP\_GenericConstant}$  (Section 26.12.1).*

### 25.3. Parallel composition

**Definition 25.6**

Given  $n$  DPIs

$$\llbracket \mathbf{d}_k : \mathbf{F}_k \rightarrow_{\text{DPI}} \mathbf{R}_k \{\mathcal{B}_k\} \rrbracket \quad (34)$$

the parallel composition

$$\text{DP\_C\_Parallel}(\llbracket \mathbf{d}_k \rrbracket) : \text{P\_C\_Product}(\llbracket \mathbf{F}_k \rrbracket) \rightarrow_{\text{DPI}} \text{P\_C\_Product}(\llbracket \mathbf{R}_k \rrbracket) \{\text{P\_C\_Product}(\llbracket \mathcal{B}_k \rrbracket)\} \quad (35)$$

is a DPI that consists of:

$$\mathbf{I} = \text{P\_C\_Product}(\llbracket \mathbf{I}_k \rrbracket) \quad (36)$$

$$\text{prov} = \text{M\_C\_Parallel}(\llbracket \text{prov}_k \rrbracket) \quad (37)$$

$$\text{req} = \text{M\_C\_Parallel}(\llbracket \text{req}_k \rrbracket) \quad (38)$$

$$\text{avail} = \text{M\_C\_CodMeet}_{\text{Bool}}(\text{M\_C\_Parallel}(\llbracket \text{avail}_k \rrbracket)) \quad (39)$$

$$\text{feas} = \text{M\_C\_CodMeet}_{\text{Bool}}(\text{M\_C\_Parallel}(\llbracket \text{feas}_k \rrbracket)) \quad (40)$$

$$\mathbf{IB} = \text{M\_C\_Parallel}(\llbracket \mathbf{IB}_k \rrbracket) \quad (41)$$

*This construction is described by the schema `DP_C_Parallel` (Section 26.12.10).*

**Lemma 25.7** (Query functoriality for the parallel composition).

$$\text{FR}(\text{DP\_C\_Parallel}(\llbracket \mathbf{d}_k \rrbracket)) = \text{U1\_C\_Parallel}(\llbracket \text{FR } \mathbf{d}_k \rrbracket) \quad (42)$$

$$\text{FRI}(\text{DP\_C\_Parallel}(\llbracket \mathbf{d}_k \rrbracket)) = \text{U\_C\_Parallel}(\llbracket \text{FRI } \mathbf{d}_k \rrbracket) \quad (43)$$

$$\text{FRB}(\text{DP\_C\_Parallel}(\llbracket \mathbf{d}_k \rrbracket)) = \text{U\_C\_Parallel}(\llbracket \text{FRB } \mathbf{d}_k \rrbracket) \quad (44)$$

$$\text{RF}(\text{DP\_C\_Parallel}(\llbracket \mathbf{d}_k \rrbracket)) = \text{L1\_C\_Parallel}(\llbracket \text{RF } \mathbf{d}_k \rrbracket) \quad (45)$$

$$\text{RFI}(\text{DP\_C\_Parallel}(\llbracket \mathbf{d}_k \rrbracket)) = \text{L\_C\_Parallel}(\llbracket \text{RFI } \mathbf{d}_k \rrbracket) \quad (46)$$

$$\text{RFB}(\text{DP\_C\_Parallel}(\llbracket \mathbf{d}_k \rrbracket)) = \text{L\_C\_Parallel}(\llbracket \text{RFB } \mathbf{d}_k \rrbracket) \quad (47)$$

**Lemma 25.8** (Approximation results for the parallel composition). For  $? \in \{\checkmark, \ominus, \odot, \star\}$ ,

$$\frac{u_k \in (\text{FR}_f^? \mathbf{d}_k)}{\text{U1\_C\_Parallel}(\llbracket u_k \rrbracket) \in \text{FR}_f^?(\text{DP\_C\_Parallel}(\llbracket \mathbf{d}_k \rrbracket))} \quad (48)$$

$$\frac{u_k \in (\text{FRI}_f^? \mathbf{d}_k)}{\text{U\_C\_Parallel}(\llbracket u_k \rrbracket) \in \text{FRI}_f^?(\text{DP\_C\_Parallel}(\llbracket \mathbf{d}_k \rrbracket))} \quad (49)$$

$$\frac{u_k \in (\text{FRB}_f^? \mathbf{d}_k)}{\text{U\_C\_Parallel}(\llbracket u_k \rrbracket) \in \text{FRB}_f^?(\text{DP\_C\_Parallel}(\llbracket \mathbf{d}_k \rrbracket))} \quad (50)$$

$$\frac{\ell_k \in (\text{RF}_f^? \mathbf{d}_k)}{\text{L1\_C\_Parallel}(\llbracket \ell_k \rrbracket) \in \text{RF}_f^?(\text{DP\_C\_Parallel}(\llbracket \mathbf{d}_k \rrbracket))} \quad (51)$$

$$\frac{\ell_k \in (\text{RFI}_f^? \mathbf{d}_k)}{\text{L\_C\_Parallel}(\llbracket \ell_k \rrbracket) \in \text{RFI}_f^?(\text{DP\_C\_Parallel}(\llbracket \mathbf{d}_k \rrbracket))} \quad (52)$$

$$\frac{\ell_k \in (\text{RFB}_f^? \mathbf{d}_k)}{\text{L\_C\_Parallel}(\llbracket \ell_k \rrbracket) \in \text{RFB}_f^?(\text{DP\_C\_Parallel}(\llbracket \mathbf{d}_k \rrbracket))} \quad (53)$$

## 25.4. Series

Here is one way to define the series of two DPIs.

**Definition 25.9** (Binary series)

Given two DPIs  $\mathbf{d}_1$  and  $\mathbf{d}_2$  such that  $\mathbf{R}_1$  and  $\mathbf{F}_2$  are subposets of a poset  $\mathbf{P}$ , the *series* of  $\mathbf{d}_1$  and  $\mathbf{d}_2$  is the DPI

$$\mathbf{d}_1 \mathbin{\text{\textcolor{brown}{;}}} \mathbf{d}_2 : \mathbf{F}_1 \rightarrow_{\mathbf{DP1}} \mathbf{R}_n \{\mathcal{B}_1 \times \mathcal{B}_2\} \quad (54)$$

defined by

$$\mathbf{I} = \text{P\_C\_Product}(\llbracket \mathbf{I}_1, \mathbf{I}_2 \rrbracket) \quad (55)$$

$$\text{prov} : \langle i_1, i_2 \rangle \mapsto \text{prov}_1(i_1) \quad (56)$$

$$\text{req} : \langle i_1, i_2 \rangle \mapsto \text{req}_2(i_2) \quad (57)$$

$$\text{avail} : \langle i_1, i_2 \rangle \mapsto \text{avail}_1(i_1) \wedge \text{avail}_2(i_2) \quad (58)$$

$$\text{IB} : \langle i_1, i_2 \rangle \mapsto \langle \text{IB}_1(i_1), \text{IB}_2(i_2) \rangle \quad (59)$$

$$\text{feas} : \langle i_1, i_2 \rangle \mapsto \text{feas}_1(i_1) \wedge \text{feas}_2(i_2) \wedge (\text{req}_1(i_1) \leq_{\mathbf{P}} \text{prov}_2(i_2)) \quad (60)$$

This construction is not associative.

The following is a way to define the n-ary series of a list of DPIs in a way that is associative.

**Definition 25.10** (n-ary series)

Given a list of DPIs  $\llbracket \mathbf{d}_k \rrbracket$ , such that for all  $k = 1 \dots n-1$ ,  $\mathbf{R}_k$  and  $\mathbf{F}_{k+1}$  are subposets of a poset  $\mathbf{P}_k$ , the *series*

$$\text{DP\_C\_Series}(\llbracket \mathbf{d}_k \rrbracket) : \mathbf{F}_1 \rightarrow_{\mathbf{DP1}} \mathbf{R}_n \{\text{P\_C\_ProductSmash}(\llbracket \mathcal{B}_k \rrbracket)\} \quad (61)$$

is the DPI that consists of:

$$\mathbf{I} = \text{P\_C\_ProductSmash}(\llbracket \mathbf{I}_k \rrbracket) \quad (62)$$

$$\text{prov} : [i_1 \mid \dots \mid i_n] \mapsto \text{prov}_1 i_1 \quad (63)$$

$$\text{req} : [i_1 \mid \dots \mid i_n] \mapsto \text{req}_n i_n \quad (64)$$

$$\text{IB} = \text{M\_C\_ParallelSmash}(\llbracket \text{IB}_k \rrbracket) \quad (65)$$

$$\text{avail} = \text{M\_C\_CodMeet}_{\text{Bool}}(\text{M\_C\_DomSmashCodProd}(\llbracket \text{avail}_k \rrbracket)) \quad (66)$$

$$\text{feas} : [i_1 \mid \dots \mid i_n] \mapsto \bigwedge_{k=1}^n \text{feas}_k(i_k) \wedge \bigwedge_{k=1}^{n-1} (\text{req}_k(i_k) \leq_{\mathbf{P}_k} \text{prov}_{k+1}(i_{k+1})) \quad (67)$$

*This construction is described by the schema DP\_C\_Series (Section 26.12.11).*

**Lemma 25.11.**

$$\text{FR DP\_C\_Series}(\llbracket \mathbf{d}_k \rrbracket) = \text{U1\_C\_Series}(\llbracket \text{FR } \mathbf{d}_k \rrbracket) \quad (68)$$

$$\text{FRI DP\_C\_Series}(\llbracket \mathbf{d}_k \rrbracket) = \text{U\_C\_Series}(\llbracket \text{FRI } \mathbf{d}_k \rrbracket) \quad (69)$$

$$\text{FRB DP\_C\_Series}(\llbracket \mathbf{d}_k \rrbracket) = \text{U\_C\_Series}(\llbracket \text{FRB } \mathbf{d}_k \rrbracket) \quad (70)$$

$$\text{RF DP\_C\_Series}(\llbracket \mathbf{d}_k \rrbracket) = \text{L1\_C\_Series}(\text{reversed}(\llbracket \text{RF } \mathbf{d}_k \rrbracket)) \quad (71)$$

$$\text{RFI DP\_C\_Series}(\llbracket \mathbf{d}_k \rrbracket) = \text{L\_C\_Series}(\text{reversed}(\llbracket \text{RFI } \mathbf{d}_k \rrbracket)) \quad (72)$$

$$\text{RFB DP\_C\_Series}(\llbracket \mathbf{d}_k \rrbracket) = \text{L\_C\_Series}(\text{reversed}(\llbracket \text{RFB } \mathbf{d}_k \rrbracket)) \quad (73)$$

**Lemma 25.12** (Approximation results for the series). For  $? \in \{\checkmark, \odot, \ominus, \star\}$ ,

$$\frac{u_k \in (\text{FR}_f^? \mathbf{d}_k)}{\text{U1\_C\_Series}(\llbracket u_k \rrbracket) \in \text{FR}_f^?(\text{DP\_C\_Series}(\llbracket \mathbf{d}_k \rrbracket))} \quad (74)$$

$$\frac{u_k \in (\text{FRI}_f^? \mathbf{d}_k)}{\text{U\_Series}(\llbracket u_k \rrbracket) \in \text{FRI}_f^?(\text{DP\_C\_Series}(\llbracket \mathbf{d}_k \rrbracket))} \quad (75)$$

$$\frac{u_k \in (\text{FRB}_f^? \mathbf{d}_k)}{\text{U\_Series}(\llbracket u_k \rrbracket) \in \text{FRB}_f^?( \text{DP\_C\_Series}(\llbracket \mathbf{d}_k \rrbracket))} \quad (76)$$

$$\frac{\ell_k \in (\text{RF}_f^? \mathbf{d}_k)}{\text{L1\_C\_Series}(\text{reversed}(\llbracket \ell_k \rrbracket)) \in \text{RF}_f^?( \text{DP\_C\_Series}(\llbracket \mathbf{d}_k \rrbracket))} \quad (77)$$

$$\frac{\ell_k \in (\text{RFI}_f^? \mathbf{d}_k)}{\text{L\_Series}(\text{reversed}(\llbracket \ell_k \rrbracket)) \in \text{RFI}_f^?( \text{DP\_C\_Series}(\llbracket \mathbf{d}_k \rrbracket))} \quad (78)$$

$$\frac{\ell_k \in (\text{RFB}_f^? \mathbf{d}_k)}{\text{L\_Series}(\text{reversed}(\llbracket \ell_k \rrbracket)) \in \text{RFB}_f^?( \text{DP\_C\_Series}(\llbracket \mathbf{d}_k \rrbracket))} \quad (79)$$

## 25.5. Intersection

### Definition 25.13

Given  $n$  DPIs

$$\llbracket \mathbf{d}_k : \mathbf{F}_k \rightarrow_{\text{DPI}} \mathbf{R}_k \{\mathcal{B}_k\} \rrbracket \quad (80)$$

whose  $\mathbf{F}_k$  are subposets of a meet-semilattice  $\bar{\mathbf{F}}$  and  $\mathbf{R}_k$  are subposets of a join-semilattice  $\bar{\mathbf{R}}$ , their intersection

$$\text{DP\_C\_Intersection}(\llbracket \mathbf{d}_k \rrbracket) : \bar{\mathbf{F}} \rightarrow_{\text{DPI}} \bar{\mathbf{R}} \{ \text{P\_C\_ProductSmash}(\llbracket \mathcal{B}_k \rrbracket) \} \quad (81)$$

is the DPI that consists of:

$$\mathbf{I} = \text{P\_C\_ProductSmash}(\llbracket \mathbf{I}_k \rrbracket) \quad (82)$$

$$\text{prov} = \text{M\_C\_CodMeet}_{\bar{\mathbf{F}}}(\text{M\_C\_DomSmashCodProd}(\llbracket \text{prov}_k \rrbracket)) \quad (83)$$

$$\text{req} = \text{M\_C\_CodJoin}_{\bar{\mathbf{R}}}(\text{M\_C\_DomSmashCodProd}(\llbracket \text{req}_k \rrbracket)) \quad (84)$$

$$\text{avail} = \text{M\_C\_CodMeet}_{\text{Bool}}(\text{M\_C\_DomSmashCodProd}(\llbracket \text{avail}_k \rrbracket)) \quad (85)$$

$$\text{feas} = \text{M\_C\_CodMeet}_{\text{Bool}}(\text{M\_C\_DomSmashCodProd}(\llbracket \text{feas}_k \rrbracket)) \quad (86)$$

$$\text{IB} = \text{M\_C\_ParallelSmash}(\llbracket \text{IB}_k \rrbracket) \quad (87)$$

This construction is described by the schema  $\text{DP\_C\_Intersection}$  (Section 26.12.12).

**Lemma 25.14** (Query functoriality for the intersection).

$$\text{FR DP\_C\_Intersection}(\llbracket \mathbf{d}_k \rrbracket) = \text{U1\_C\_Intersection}(\llbracket \text{FR } \mathbf{d}_k \rrbracket) \quad (88)$$

$$\text{FRI DP\_C\_Intersection}(\llbracket \mathbf{d}_k \rrbracket) = \text{U\_C\_Intersection}(\llbracket \text{FRI } \mathbf{d}_k \rrbracket) \quad (89)$$

$$\text{FRB DP\_C\_Intersection}(\llbracket \mathbf{d}_k \rrbracket) = \text{U\_C\_Intersection}(\llbracket \text{FRB } \mathbf{d}_k \rrbracket) \quad (90)$$

$$\text{RF DP\_C\_Intersection}(\llbracket \mathbf{d}_k \rrbracket) = \text{L1\_C\_Intersection}(\llbracket \text{RF } \mathbf{d}_k \rrbracket) \quad (91)$$

$$\text{RFI DP\_C\_Intersection}(\llbracket \mathbf{d}_k \rrbracket) = \text{L\_C\_Intersection}(\llbracket \text{RFI } \mathbf{d}_k \rrbracket) \quad (92)$$

$$\text{RFB DP\_C\_Intersection}(\llbracket \mathbf{d}_k \rrbracket) = \text{L\_C\_Intersection}(\llbracket \text{RFB } \mathbf{d}_k \rrbracket) \quad (93)$$

**Lemma 25.15** (Approximation results for the intersection). For  $? \in \{\checkmark, \ominus, \odot, \star\}$ ,

$$\frac{u_k \in (\text{FR}_f^? \mathbf{d}_k)}{\text{U1\_C\_Intersection}(\llbracket u_k \rrbracket) \in \text{FR}_f^?( \text{DP\_C\_Intersection}(\llbracket \mathbf{d}_k \rrbracket))} \quad (94)$$

$$\frac{u_k \in (\text{FRI}_f^? \mathbf{d}_k)}{\text{U\_C\_Intersection}(\llbracket u_k \rrbracket) \in \text{FRI}_f^?( \text{DP\_C\_Intersection}(\llbracket \mathbf{d}_k \rrbracket))} \quad (95)$$



$$\frac{u_k \in (\text{FRB}_f^? \mathbf{d}_k)}{\text{U\_C\_Intersection}(\llbracket u_k \rrbracket) \in \text{FRB}_f^?(\text{DP\_C\_Intersection}(\llbracket \mathbf{d}_k \rrbracket))} \quad (96)$$

$$\frac{\ell_k \in (\text{RF}_f^? \mathbf{d}_k)}{\text{L1\_C\_Intersection}(\llbracket \ell_k \rrbracket) \in \text{RF}_f^?(\text{DP\_C\_Intersection}(\llbracket \mathbf{d}_k \rrbracket))} \quad (97)$$

$$\frac{\ell_k \in (\text{RFI}_f^? \mathbf{d}_k)}{\text{L\_C\_Intersection}(\llbracket \ell_k \rrbracket) \in \text{RFI}_f^?(\text{DP\_C\_Intersection}(\llbracket \mathbf{d}_k \rrbracket))} \quad (98)$$

$$\frac{\ell_k \in (\text{RFB}_f^? \mathbf{d}_k)}{\text{L\_C\_Intersection}(\llbracket \ell_k \rrbracket) \in \text{RFB}_f^?(\text{DP\_C\_Intersection}(\llbracket \mathbf{d}_k \rrbracket))} \quad (99)$$

## 25.6. Union

### Definition 25.16

Given  $n$  DPIs  $\llbracket \mathbf{d}_k : \mathbf{F}_k \rightarrow_{\text{DPI}} \mathbf{R}_k \{\mathcal{B}_k\} \rrbracket$  whose  $\mathbf{F}_k$  are subposets of a poset  $\bar{\mathbf{F}}$  and  $\mathbf{R}_k$  are subposets of a poset  $\bar{\mathbf{R}}$ , their union

$$\text{DP\_C\_Union}(\llbracket \mathbf{d}_k \rrbracket) : \bar{\mathbf{F}} \rightarrow_{\text{DPI}} \bar{\mathbf{R}} \{\text{P\_C\_SumSmash}(\llbracket \mathcal{B}_k \rrbracket)\} \quad (100)$$

is the DPI that consists of:

$$\mathbf{I} = \text{P\_C\_SumSmash}(\llbracket \mathbf{I}_k \rrbracket) \quad (101)$$

$$\text{prov} = \text{M\_C\_CoproductSmash}(\llbracket \text{prov}_k \rrbracket) \quad (102)$$

$$\text{req} = \text{M\_C\_CoproductSmash}(\llbracket \text{req}_k \rrbracket) \quad (103)$$

$$\text{avail} = \text{M\_C\_CoproductSmash}(\llbracket \text{avail}_k \rrbracket) \quad (104)$$

$$\text{feas} = \text{M\_C\_CoproductSmash}(\llbracket \text{feas}_k \rrbracket) \quad (105)$$

$$\mathbf{IB} = \text{M\_C\_SumSmash}(\llbracket \mathbf{IB}_k \rrbracket) \quad (106)$$

This construction is described by the schema  $\text{DP\_C\_Union}$  (Section 26.12.13).

**Lemma 25.17** (Query functoriality for the union).

$$\text{FR DP\_C\_Union}(\llbracket \mathbf{d}_k \rrbracket) = \text{U1\_C\_Union}(\llbracket \text{FR } \mathbf{d}_{k,k} \rrbracket) \quad (107)$$

$$\text{FRI DP\_C\_Union}(\llbracket \mathbf{d}_k \rrbracket) = \text{U\_C\_Union}(\llbracket \text{FRI } \mathbf{d}_{k,k} \rrbracket) \quad (108)$$

$$\text{FRB DP\_C\_Union}(\llbracket \mathbf{d}_k \rrbracket) = \text{U\_C\_Union}(\llbracket \text{FRB } \mathbf{d}_{k,k} \rrbracket) \quad (109)$$

$$\text{RF DP\_C\_Union}(\llbracket \mathbf{d}_k \rrbracket) = \text{L1\_C\_Union}(\llbracket \text{RF } \mathbf{d}_k \rrbracket) \quad (110)$$

$$\text{RFI DP\_C\_Union}(\llbracket \mathbf{d}_k \rrbracket) = \text{L\_C\_Union}(\llbracket \text{RFI } \mathbf{d}_k \rrbracket) \quad (111)$$

$$\text{RFB DP\_C\_Union}(\llbracket \mathbf{d}_k \rrbracket) = \text{L\_C\_Union}(\llbracket \text{RFB } \mathbf{d}_k \rrbracket) \quad (112)$$

**Lemma 25.18** (Approximation results for the union). For  $? \in \{\checkmark, \ominus, \odot, \star\}$ ,

$$\frac{u_k \in (\text{FR}_f^? \mathbf{d}_k)}{\text{U1\_C\_Union}(\llbracket u_k \rrbracket) \in \text{FR}_f^?(\text{DP\_C\_Union}(\llbracket \mathbf{d}_k \rrbracket))} \quad (113)$$

$$\frac{u_k \in (\text{FRI}_f^? \mathbf{d}_k)}{\text{U\_C\_Union}(\llbracket u_k \rrbracket) \in \text{FRI}_f^?(\text{DP\_C\_Union}(\llbracket \mathbf{d}_k \rrbracket))} \quad (114)$$

$$\frac{u_k \in (\text{FRB}_f^? \mathbf{d}_k)}{\text{U\_C\_Union}(\llbracket u_k \rrbracket) \in \text{FRB}_f^?(\text{DP\_C\_Union}(\llbracket \mathbf{d}_k \rrbracket))} \quad (115)$$

$$\frac{\ell_k \in (\text{RF}_f^? \mathbf{d}_k)}{\text{L1\_C\_Union}(\llbracket \ell_k \rrbracket) \in \text{RF}_f^?(\text{DP\_C\_Union}(\llbracket \mathbf{d}_k \rrbracket))} \quad (116)$$

$$\frac{\ell_k \in (\text{RFI}_f^? \mathbf{d}_k)}{\text{L\_C\_Union}(\llbracket \ell_k \rrbracket) \in \text{RFI}_f^?(\text{DP\_C\_Union}(\llbracket \mathbf{d}_k \rrbracket))} \quad (117)$$

$$\frac{\ell_k \in (\text{RFB}_f^? \mathbf{d}_k)}{\text{L\_C\_Union}(\llbracket \ell_k \rrbracket) \in \text{RFB}_f^?(\text{DP\_C\_Union}(\llbracket \mathbf{d}_k \rrbracket))} \quad (118)$$

## 25.7. Trace

### Definition 25.19

Given a DPI

$$\mathbf{d}_0 : \text{P\_C\_Product}(\llbracket \mathbf{F}_1, \mathbf{F}_2 \rrbracket) \rightarrow_{\text{DPI}} \text{P\_C\_Product}(\llbracket \mathbf{R}_1, \mathbf{R}_2 \rrbracket) \{\mathcal{B}_0\} \quad (119)$$

such that  $\mathbf{R}_2$  and  $\mathbf{F}_2$  are subposets of a poset  $\mathbf{P}$ , the *trace* of  $\mathbf{d}_0$  is the DPI

$$\text{DP\_C\_Trace}(\mathbf{d}_0) : \mathbf{F}_1 \rightarrow_{\text{DPI}} \mathbf{R}_1 \{\mathcal{B}_0\} \quad (120)$$

whose data is:

$$\mathbf{I} = \mathbf{I}_0 \quad (121)$$

$$\mathbf{IB} = \mathbf{IB}_0 \quad (122)$$

$$\text{prov} = \text{prov}_0 \circ \pi_1 \quad (123)$$

$$\text{req} = \text{req}_0 \circ \pi_1 \quad (124)$$

$$\text{avail} = \text{avail}_0 \quad (125)$$

$$\text{feas} : i \mapsto \text{feas}_0(i) \wedge ([\text{req}_0 \circ \pi_2](i) \leq_{\mathbf{P}} [\text{prov}_0 \circ \pi_2](i)) \quad (126)$$

This construction is described by the schema  $\text{DP\_C\_Trace}$  (Section 26.12.14).

**Lemma 25.20** (Query functoriality for the trace).

$$\text{FR DP\_C\_Trace}(\mathbf{d}_0) = \text{U1\_C\_Trace}(\text{FR } \mathbf{d}_0) \quad (127)$$

$$\text{FRI DP\_C\_Trace}(\mathbf{d}_0) = \text{U\_C\_Trace}(\text{FRI } \mathbf{d}_0) \quad (128)$$

$$\text{FRB DP\_C\_Trace}(\mathbf{d}_0) = \text{U\_C\_Trace}(\text{FRB } \mathbf{d}_0) \quad (129)$$

$$\text{RF DP\_C\_Trace}(\mathbf{d}_0) = \text{L1\_C\_Trace}(\text{RF } \mathbf{d}_0) \quad (130)$$

$$\text{RFI DP\_C\_Trace}(\mathbf{d}_0) = \text{L\_C\_Trace}(\text{RFI } \mathbf{d}_0) \quad (131)$$

$$\text{RFB DP\_C\_Trace}(\mathbf{d}_0) = \text{L\_C\_Trace}(\text{RFB } \mathbf{d}_0) \quad (132)$$

**Lemma 25.21** (Approximation results for the trace). For  $? \in \{\checkmark, \ominus, \odot, \star\}$ ,

$$\frac{u_0 \in (\text{FR}_f^? \mathbf{d}_0)}{\text{U1\_C\_Trace}(u_0) \in \text{FR}_f^?(\text{DP\_C\_Trace}(\mathbf{d}_0))} \quad (133)$$

$$\frac{u_0 \in (\text{FRI}_f^? \mathbf{d}_0)}{\text{U\_C\_Trace}(u_0) \in \text{FRI}_f^?(\text{DP\_C\_Trace}(\mathbf{d}_0))} \quad (134)$$

$$\frac{u_0 \in (\text{FRB}_f^? \mathbf{d}_0)}{\text{U\_C\_Trace}(u_0) \in \text{FRB}_f^?(\text{DP\_C\_Trace}(\mathbf{d}_0))} \quad (135)$$

$$\frac{\ell_0 \in (\text{RF}_f^? \mathbf{d}_0)}{\text{L1\_C\_Trace}(\ell_0) \in \text{RF}_f^?(\text{DP\_C\_Trace}(\mathbf{d}_0))} \quad (136)$$

$$\begin{array}{c}
\ell_0 \in (\text{RFI}_i^? \mathbf{d}_0) \\
\hline
\text{L\_C\_Trace}(\ell_0) \in \text{RFI}_i^?(\text{DP\_C\_Trace}(\mathbf{d}_0))
\end{array}
\tag{137}$$

$$\begin{array}{c}
\ell_0 \in (\text{RFB}_i^? \mathbf{d}_0) \\
\hline
\text{L\_C\_Trace}(\ell_0) \in \text{RFB}_i^?(\text{DP\_C\_Trace}(\mathbf{d}_0))
\end{array}
\tag{138}$$

Part F.

MCDP Format 2

## 26. Top level

## 26.1. Root - Top-level object types for what can be serialized in a file.

Property	Symbol	Type	Description
version		string?	Version of the MCDP format used to serialize this object (major.minor).
description		string?	A human-readable description of the object used for debug purposes.
hash		string?	Unique hash for the object.
kind		string	Discriminator variable to distinguish subtypes.

The Root schema contains as subtypes all kinds of objects that can be serialized in a MCDP file during an export operation.

### Subtypes based on the value for kind

"Poset"	A poset.
"MonotoneMap"	Monotone maps
"L1Map"	Map to lower sets of functionalities.
"U1Map"	Map to upper sets of resources.
"LMap"	Map to lower sets of functionalities and implementations.
"UMap"	Map to upper sets of resources and implementations.
"SL1Map"	Scalable map to lower sets of functionalities.
"SU1Map"	Scalable map to upper sets of resources.
"SLMap"	Scalable map to lower sets of functionalities and implementations.
"SUMap"	Scalable map to upper sets of resources and implementations.
"DP"	Design problem with implementations (DPI)
"NDP"	Named DPs represent a graph of DPs with named nodes and node ports.
"NDPInterface"	The interface of a named DP.
"NDPTemplate"	A template for an NDP.
"Query"	Queries
"Value"	A typed value
"Check"	Checks for the maps, as used in test cases.

26.2. Poset - A poset.

Data

**Extends:** Root(version, description, hash, kind = "Poset")

Property	Symbol	Type	Description
address		Address?	Pointer to the entity that generated this object.
type		string	Discriminator variable to distinguish subtypes.

The Poset objects represent various kinds of posets.

Some of these are “primitive” posets, such as P\_Decimal, P\_Bool, P\_Finite.

Some of these are “composite” posets. They are indicated by a prefix P\_C\_. Some examples are P\_C\_Product and P\_C\_Sum.

The posets whose name starts with P\_F are “filters”. These represent a subposet of another poset. Some examples are P\_F\_Interval and P\_F\_Subposet.

Each poset defines also the format in which its data can be serialized in YAML/CBOR.

Subtypes based on the value for type

"P_Bool"	The poset of boolean values
"P_Decimal"	Decimal numbers with fixed precision.
"P_Finite"	Arbitrary finite poset
"P_Float"	Poset of floating point numbers.
"P_Fractions"	Fractions with a maximum absolute value for numerator and denominator.
"P_Integer"	Poset of integers.
"P_Unknown"	Placeholder for an unknown poset
"P_C_Arrow"	Arrow constructors for posets.
"P_C_Discretized"	Discretized version of a poset.
"P_C_LowerSets"	The poset of lower sets of a given poset.
"P_C_Opposite"	Opposite of a poset
"P_C_Power"	Power poset of a given poset.
"P_C_Twisted"	Twisted arrow construction of a poset.
"P_C_Units"	A poset with units
"P_C_UpperSets"	The poset of upper sets of a given poset.
"P_C_Lexicographic"	Lexicographic product of posets
"P_C_Product"	Cartesian product of posets
"P_C_ProductSmash"	Poset smash product
"P_C_Sum"	Direct sum of posets.
"P_C_SumSmash"	Direct (smash) sum of posets
"P_F_Bounded"	A subposet that allows to sample a numeric poset.
"P_F_C_Intersection"	Intersection of posets.
"P_F_C_Union"	Union of posets
"P_F_Interval"	An interval in a poset.
"P_F_LowerClosure"	Lower closure in a poset.
"P_F_Subposet"	A finite subposet of an ambient poset.
"P_F_UpperClosure"	Upper closure in a poset.

26.2.1. P\_Bool - The poset of boolean values

Data

**Extends:** Poset(address, type = "P\_Bool")

Examples

```
{kind: Poset, type: P_Bool}
```

This is how to define the poset. There are no other parameters.

*Examples of values serialization:*

```
true
```

```
false
```

### 26.2.2. P\_Decimal - Decimal numbers with fixed precision.

Data

Extends: Poset(address, type = "P\_Decimal")

Property	Symbol	Type	Description
precision		integer	Number of decimal places.

Examples

```
{kind: Poset, type: P_Decimal, precision: 9}
```

Examples of values serialization:

```
"0.0"
```

The value 0.0.

```
"+inf"
```

```
"-inf"
```

### 26.2.3. P\_Finite - Arbitrary finite poset

Data	<b>Extends:</b> Poset(address, type = "P_Finite")			
	<b>Property</b>	<b>Symbol</b>	<b>Type</b>	<b>Description</b>
	elements		array[string]	The elements of the poset, strings.
	relations		array[array[string]]	The relations of the poset, each relation is a pair of elements. The first element is less than the second element.
	aliases		dict[string,array[string]]?	Aliases for the elements of the poset. The keys are the aliases, and the values are arrays of elements that are equivalent to the alias.

A finite poset is a set of elements with a partial order defined by a set of relations.

Examples	<pre>kind: Poset type: P_Finite elements: [] relations: [] aliases: {}</pre>		This is the empty poset.	
	<pre>kind: Poset type: P_Finite elements: [a, b] relations: []</pre>		This is a poset with two elements that are not related.	
	<pre>kind: Poset type: P_Finite elements: [a, b] relations: [[a, b]]</pre>		This is a poset with two elements with $a \leq b$ .	
	<pre>kind: Poset type: P_Finite elements: [a, b, c] relations: [[a, b], [b, c]] aliases: {a: [a1, a2]}</pre>		<p>This is a pre-order with 5 elements: a, b, c, a1, and a2. The elements a, a1, a2 are equivalent. The element a is used as the representative of the equivalence class.</p> <p><i>Examples of values serialization:</i></p> <pre>a</pre> <pre>a2</pre>	



#### 26.2.4. P\_Float - Poset of floating point numbers.

Data	<b>Extends:</b> Poset(address, type = "P_Float")									
	<table><tr><th>Property</th><th>Symbol</th><th>Type</th><th>Description</th></tr><tr><td>size</td><td></td><td>string</td><td>Precision of the floating point number. Current supported values are f32 and f64. <i>Possible values:</i> "f8", "f16", "f32", "f64", "f80", "f128"</td></tr></table>	Property	Symbol	Type	Description	size		string	Precision of the floating point number. Current supported values are f32 and f64. <i>Possible values:</i> "f8", "f16", "f32", "f64", "f80", "f128"	
Property	Symbol	Type	Description							
size		string	Precision of the floating point number. Current supported values are f32 and f64. <i>Possible values:</i> "f8", "f16", "f32", "f64", "f80", "f128"							
Examples	<pre>{kind: Poset, type: P_Float, size: f32}</pre>									

#### 26.2.5. P\_Fractions - Fractions with a maximum absolute value for numerator and denominator.

Data

Extends: Poset(address, type = "P\_Fractions")

Property	Symbol	Type	Description
size		string	Precision of the fraction. <i>Possible values:</i> "i8", "i16", "i32", "i64", "i128"
max_abs_numerator		integer	Maximum absolute value for the numerator.
max_abs_denominator		integer	Maximum absolute value for the denominator.

Fractions with a maximum absolute value for numerator and denominator.

Examples

```
kind: Poset
type: P_Fractions
size: i32
max_abs_numerator: 1000
max_abs_denominator: 1000
```

Examples of values serialization:

"0"

"1/0"

"-1/0"

#### 26.2.6. P\_Integer - Poset of integers.

Data	<b>Extends:</b> Poset(address, type = "P_Integer")									
	<table><tr><th>Property</th><th>Symbol</th><th>Type</th><th>Description</th></tr><tr><td>size</td><td></td><td>string</td><td>Bit size of the integer. <i>Possible values:</i> "i8", "i16", "i32", "i64", "i128"</td></tr></table>	Property	Symbol	Type	Description	size		string	Bit size of the integer. <i>Possible values:</i> "i8", "i16", "i32", "i64", "i128"	
Property	Symbol	Type	Description							
size		string	Bit size of the integer. <i>Possible values:</i> "i8", "i16", "i32", "i64", "i128"							
Examples	<div><pre>{kind: Poset, type: P_Integer, size: i32}</pre></div>									
	<div><div><i>Examples of values serialization:</i></div><div><div>"0"</div><div>The value 0.</div></div><div><div>"10000000000000000"</div><div>A large positive integer.</div></div></div>									

26.2.7. P\_Unknown - Placeholder for an unknown poset

Data

**Extends:** Poset(address, type = "P\_Unknown")

Examples

```
{kind: Poset, type: P_Unknown}
```

26.2.8. P\_C\_Arrow - Arrow constructors for posets.

Data

**Extends:** Poset(address, type = "P\_C\_Arrow")

Property	Symbol	Type	Description
poset		Poset	The base poset.

Discussed in Section 17.2.2 (Arrow constructions)

The arrow construction of a poset is a poset where the values are pairs of elements from the underlying poset.  
This poset has the same elements as P\_C\_Twisted, but the order is different.

Examples

```
kind: Poset
type: P_C_Arrow
poset: {kind: Poset, type: P_Decimal}
```

Poset of intervals of decimal numbers.  
*Examples of values serialization:*

```
["0", "10"]
```

The interval from 0 to 10, inclusive.

26.2.9. P\_C\_Discretized - Discretized version of a poset.

Data

**Extends:** Poset(address, type = "P\_C\_Discretized")

Property	Symbol	Type	Description
poset		Poset	The base poset.

Discussed in Section 17.2.3 (Discretized version of a poset)

Examples

```
kind: Poset
type: P_C_Discretized
poset: {kind: Poset, type: P_Bool}
```

The poset of boolean values without the order

26.2.10. P\_C\_LowerSets - The poset of lower sets of a given poset.

Data

Extends: Poset(address, type = "P\_C\_LowerSets")

Property	Symbol	Type	Description
poset		Poset	The base poset.

Discussed in Section 17.2.4 (Posets of subsets)

Examples

<pre>kind: Poset type: P_C_LowerSets poset: {kind: Poset, type: P_Decimal}</pre>	Poset of lower sets of decimal numbers. Examples of values serialization:  ["10"] The lower set ↓ 10.
--	---

26.2.11. P\_C\_Opposite - Opposite of a poset

Data

Extends: Poset(address, type = "P\_C\_Opposite")

Property	Symbol	Type	Description
poset		Poset	The base poset.

Discussed in Section 17.2.1 (Opposite of a poset)

Examples

<pre>kind: Poset type: P_C_Opposite poset: {kind: Poset, type: P_Decimal}</pre>	The opposite poset of decimal numbers.
---	--

### 26.2.12. P\_C\_Power - Power poset of a given poset.

Data

**Extends:** Poset(address, type = "P\_C\_Power")

Property	Symbol	Type	Description
poset		Poset	The base poset.

*Discussed in Section 17.2.4 (Posets of subsets)*

Examples

```
kind: Poset
type: P_C_Power
poset: {kind: Poset, type: P_Decimal}
```

Poset of powersets of decimal numbers.

*Examples of values serialization:*

```
["10", "20"]
```

The set {10, 20}.

```
[]
```

The empty subset.

### 26.2.13. P\_C\_Twisted - Twisted arrow construction of a poset.

Data

**Extends:** Poset(address, type = "P\_C\_Twisted")

Property	Symbol	Type	Description
poset		Poset	The base poset.

*Discussed in Section 17.2.2 (Arrow constructions)*

The twisted arrow construction of a poset is a poset where the values are pairs of elements from the underlying poset.

This poset has the same elements as P\_C\_Arrow, but the order is different.

Examples

```
kind: Poset
type: P_C_Twisted
poset: {kind: Poset, type: P_Decimal}
```

Poset of intervals of decimal numbers.

*Examples of values serialization:*

```
["0", "10"]
```

The interval from 0 to 10, inclusive.

### 26.2.14. P\_C\_Units - A poset with units

Data

**Extends:** Poset(address, type = "P\_C\_Units")

Property	Symbol	Type	Description
poset		Poset	The base poset.
units		Unit	The units of the poset.

```

kind: Poset
type: P_C_Units
poset: {kind: Poset, type: P_Decimal}
units: {kind: Unit, type: Unit_Single, units: m^2/s}

```

The poset of decimal numbers with units of  $m^2/s$ .

```

kind: Poset
type: P_C_Units
poset:
  kind: Poset
  type: P_C_Product
  subs:
    - {kind: Poset, type: P_Decimal}
    - {kind: Poset, type: P_Decimal}
units:
  kind: Unit
  type: Unit_Vector
  subs:
    - {kind: Unit, type: Unit_Single, units: m}
    - {kind: Unit, type: Unit_Single, units: g}

```

This is how to assign the units m (meters) and g (grams) to a product  $P \times Q$ .

### 26.2.15. P\_C\_UpperSets - The poset of upper sets of a given poset.

**Extends:** Poset(address, type = "P\_C\_UpperSets")

Property	Symbol	Type	Description
poset		Poset	The base poset.

*Discussed in Section 17.2.4 (Posets of subsets)*

```

kind: Poset
type: P_C_UpperSets
poset: {kind: Poset, type: P_Decimal}

```

Poset of uppersets of decimal numbers.

*Examples of values serialization:*

```
["10"]
```

The upper set  $\uparrow 10$ .

### 26.2.16. P\_C\_Lexicographic - Lexicographic product of posets

**Extends:** Poset(address, type = "P\_C\_Lexicographic")

Property	Symbol	Type	Description
subs		array[Poset]	A list of posets that are composed together.
labels		array[string]?	A list of labels for the posets.

*Discussed in Section 17.3.3 (Lexicographic product of posets)*

This is the lexicographic product of posets. The order is defined by the lexicographic order of the elements of the subs posets.

The elements are the same as in P\_C\_Product, but the order is different.

```

kind: Poset
type: P_C_Lexicographic
subs:
  - {kind: Poset, type: P_Decimal}
  - {kind: Poset, type: P_Bool}

```

### 26.2.17. P\_C\_Product - Cartesian product of posets

Data

**Extends:** Poset(address, type = "P\_C\_Product")

Property	Symbol	Type	Description
subs		array[Poset]	A list of posets that are composed together.
labels		array[string]?	A list of labels for the posets.

*Discussed in Section 17.3.1 (Cartesian product of posets)*

This is the Cartesian product of posets.

Examples

```
kind: Poset
type: P_C_Product
subs:
  - {kind: Poset, type: P_Decimal}
  - {kind: Poset, type: P_Bool}
```

The product of a decimal poset and a boolean poset.

*Examples of values serialization:*

```
["10", true]
```

The element  $\langle 10, T \rangle$ .

```
{kind: Poset, type: P_C_Product, subs: []}
```

This is the empty product. It has exactly one element, the empty tuple.

*Examples of values serialization:*

```
[]
```

The empty tuple

### 26.2.18. P\_C\_ProductSmash - Poset smash product

Data

**Extends:** Poset(address, type = "P\_C\_ProductSmash")

Property	Symbol	Type	Description
subs		array[Poset]	A list of posets that are composed together.
labels		array[string]?	A list of labels for the posets.
ranges		array[Range]	The ranges of the posets in the smash product. See also P_C_Sum
naked		array[boolean]	Whether each poset is "naked" or not.

*Discussed in Section 17.3.1 (Smash product of posets)*

This poset is the “smash product”. Its elements are concatenation of tuples of elements of the underlying posets.

We say that a subposet is “naked” if it is not of a “smash nature” and its elements need to be wrapped as a tuple.

The posets that are of a smash nature are:

- Other instances of P\_C\_ProductSmash
- Instances of P\_C\_SumSmash
- Instances of filters (e.g. P\_F\_Subposet or P\_C\_Units) whose underlying poset is of smash nature.

The ranges parameter describe what is the range in the tuple of each subposet.

Both the ranges and naked parameters are redundant in the sense that they can be computed from the subs, but they are provided for convenience and to avoid recomputing them.

Examples

```

kind: Poset
type: P_C_ProductSmash
subs: []
naked: []
ranges: []

```

The empty smash product. It has exactly one element, the empty tuple.

*Examples of values serialization:*

```
[]
```

The empty tuple

```

kind: Poset
type: P_C_ProductSmash
subs:
  - {kind: Poset, type: P_Decimal}
  - {kind: Poset, type: P_Bool}
  - {kind: Poset, type: P_C_ProductSmash}
    subs:
      - {kind: Poset, type: P_Decimal}
      - {kind: Poset, type: P_Bool}
    naked: [true, true]
    ranges:
      - {type: Range, start: 0, stop: 1, ntot: 2}
      - {type: Range, start: 1, stop: 2, ntot: 2}
  - {kind: Poset, type: P_Decimal}
naked: [true, false, true]
ranges:
  - {type: Range, start: 0, stop: 1, ntot: 4}
  - {type: Range, start: 1, stop: 3, ntot: 4}
  - {type: Range, start: 3, stop: 4, ntot: 4}

```

This is the smash product of 3 posets: booleans, a smash product of two posets, and the decimals. The first and the last posets are naked, while the middle one is not because it is of a smash nature.

*Examples of values serialization:*

```
[true, "10", false, "20"]
```

### 26.2.19. P\_C\_Sum - Direct sum of posets.

**Extends:** Poset(address, type = "P\_C\_Sum")

Data

Property	Symbol	Type	Description
subs		array[Poset]	A list of posets that are composed together.
labels		array[string]?	A list of labels for the posets.

*Discussed in Section 17.3.2 (Direct sum of posets)*

Examples

```

kind: Poset
type: P_C_Sum
subs:
  - {kind: Poset, type: P_Decimal}
  - {kind: Poset, type: P_Bool}

```

### 26.2.20. P\_C\_SumSmash - Direct (smash) sum of posets

**Extends:** Poset(address, type = "P\_C\_SumSmash")

Data

Property	Symbol	Type	Description
subs		array[Poset]	A list of posets that are composed together.
labels		array[string]?	A list of labels for the posets.
trivial		boolean	
ranges		array[Range]	
naked		array[boolean]	

*Discussed in Section 17.3.2 (Direct (smash) sum of posets)*

This is the direct sum of posets but of “smash nature”. See the description of P\_C\_ProductSmash for the description of ranges and naked.

```
kind: Poset
type: P_C_SumSmash
trivial: false
subs:
  - {kind: Poset, type: P_Decimal}
  - {kind: Poset, type: P_Bool}
naked: [true, true]
ranges:
  - {type: Range, start: 0, stop: 1, ntot: 2}
  - {type: Range, start: 0, stop: 1, ntot: 2}
```

Extends: Poset(address, type = "P\_F\_Bounded")

Property	Symbol	Type	Description
poset		Poset	The ambient poset.
bottom	B	any	The bottom element of the subposet.
top	T	any	The top element of the subposet.
step	S	string	The step size of the subposet.
offset	O	any	The offset
bound_high	H	any	An upper bound.
bound_low	L	any	A lower bound.

A bounded poset is a subposet of a numeric poset that allows to discretize it.

It is defined by 5 values that satisfy:

$$B \leq L \leq O \leq H \leq T \tag{1}$$

The subposet is defined by

$$\{B, T\} \cup ([L, H] \cap \{O + i \cdot S \mid i \in \mathbb{Z}\}) \tag{2}$$

where  $[L, H]$  is the closed interval between L and H.

```
kind: Poset
type: P_F_Bounded
poset: {kind: Poset, type: P_Decimal}
bottom: "0"
bound_low: "0"
step: "1"
offset: "0"
bound_high: "+inf"
top: "+inf"
```

This defines the nonnegative integers.

```
kind: Poset
type: P_F_Bounded
poset: {kind: Poset, type: P_Decimal}
bottom: "-inf"
bound_low: "-inf"
step: "2"
offset: "1"
bound_high: "+inf"
top: "+inf"
```

This defines the odd integers. The offset is 1 and the step is 2.

```
kind: Poset
type: P_F_Bounded
poset: {kind: Poset, type: P_Decimal}
bottom: "-inf"
bound_low: "0"
offset: "0"
step: "1.5"
bound_high: "6"
top: "+inf"
```

This complex example describes the poset with the elements  $\{-\infty, 0, 1.5, 3, 4.5, 6, +\infty\}$ .



26.2.22. P\_F\_C\_Intersection - Intersection of posets.

Data

Extends: Poset(address, type = "P_F_C_Intersection")			
Property	Symbol	Type	Description
ambient		Poset	The ambient poset that includes the others.
subs		array[Poset]	The posets that are included in the intersection. They are all subposets of the ambient poset.
labels		array[string]?	Labels for the posets.

Discussed in Section 17.4.4 (Union and Intersection of sub posets)

26.2.23. P\_F\_C\_Union - Union of posets

Data

Extends: Poset(address, type = "P_F_C_Union")			
Property	Symbol	Type	Description
ambient		Poset	The ambient poset that includes the others.
subs		array[Poset]	The posets that are included in the union. They are all subposets of the ambient poset.
labels		array[string]?	Labels for the posets.

Discussed in Section 17.4.4 (Union and Intersection of sub posets)

Examples

<pre>kind: Poset type: P_F_C_Union ambient: {kind: Poset, type: P_Decimal} subs:   - kind: Poset     type: P_F_Interval     poset: {kind: Poset, type: P_Decimal}     low: "10"     high: "20"   - kind: Poset     type: P_F_Interval     poset: {kind: Poset, type: P_Decimal}     low: "30"     high: "35"</pre>	This describes the poset $[10, 20] \cup [30, 35]$ in the ambient poset of decimal numbers.
--	--

#### 26.2.24. P\_F\_Interval - An interval in a poset.

Data	<b>Extends:</b> Poset(address, type = "P_F_Interval")			
	Property	Symbol	Type	Description
	poset	$P$	Poset	The ambient poset.
	low	$a$	any	The lower bound of the interval.
	high	$b$	any	The upper bound of the interval.

*Discussed in Section 17.4.2 (Interval in a poset)*

Let  $P$  be a poset, and let  $a$  and  $b$  be elements of  $P$ . Then this poset represents the subposet in  $P$ , given by  $\{x \in P \mid a \leq x \leq b\}$ .

Examples	<pre> kind: Poset type: P_F_Interval poset: {kind: Poset, type: P_Decimal} low: "10" high: "20" </pre>	This describes the interval $[10, 20]$ in the ambient poset of decimal numbers.
----------	--	---

#### 26.2.25. P\_F\_LowerClosure - Lower closure in a poset.

Data	<b>Extends:</b> Poset(address, type = "P_F_LowerClosure")			
	Property	Symbol	Type	Description
	poset		Poset	The ambient poset.
	ls		LowerSet	The lower set.

*Discussed in Section 17.4.3 (Lower and upper closure in a poset)*

Examples	<pre> kind: Poset type: P_F_LowerClosure poset:   kind: Poset   type: P_C_Product   subs:     - {kind: Poset, type: P_Decimal}     - {kind: Poset, type: P_Bool} ls:   kind: LowerSet   type: LowerSet_LowerClosure   points: [{"10", true}, {"20", false}] </pre>	This describes the lower closure of the set $\{(10, \top), (20, \perp)\}$ in the ambient poset of decimal numbers and booleans.
----------	--	---

#### 26.2.26. P\_F\_Subposet - A finite subposet of an ambient poset.

Data	<b>Extends:</b> Poset(address, type = "P_F_Subposet")			
	Property	Symbol	Type	Description
	elements		array[any]	The elements of the subposet.
	poset		Poset	The ambient poset that contains the elements.

*Discussed in Section 17.4.1 (Finite subposet of an ambient poset)*

A finite subposet of an ambient poset specified by a list of elements.

As a particular case, the list of elements can be empty.

```

kind: Poset
type: P_F_Subposet
poset: {kind: Poset, type: P_Decimal}
elements: ["0.1", "0.2"]

```

This describes the subposet  $\{0.1, 0.2\}$  in the ambient poset of decimal numbers.

```

kind: Poset
type: P_F_Subposet
poset: {kind: Poset, type: P_Decimal}
elements: []

```

This describes the empty subposet in the ambient poset of decimal numbers.

### 26.2.27. P\_F\_UpperClosure - Upper closure in a poset.

**Extends:** Poset(address, type = "P\_F\_UpperClosure")

Property	Symbol	Type	Description
poset		Poset	The ambient poset.
us		UpperSet	The upper set.

*Discussed in Section 17.4.3 (Lower and upper closure in a poset)*

```

kind: Poset
type: P_F_UpperClosure
poset:
  kind: Poset
  type: P_C_Product
  subs:
    - {kind: Poset, type: P_Decimal}
    - {kind: Poset, type: P_Bool}
us:
  kind: UpperSet
  type: UpperSet_UpperClosure
  points: [{"10", true}, {"20", false}]

```

This describes the upper closure of the set  $\{(10, \top), (20, \perp)\}$  in the ambient poset of decimal numbers and booleans.

## 26.3. MonotoneMap - Monotone maps

Data

**Extends:** Root(version, description, hash, kind = "MonotoneMap")

Property	Symbol	Type	Description
dom	dom	Poset	Domain of the monotone map
cod	cod	Poset	Codomain of the monotone map
type		string	Discriminator variable to distinguish subtypes.

Subtypes based on the value for type

"M_Constant"	A constant function
"M_Empty"	The unique map from the empty set to another
"M_Explicit"	A map defined pointwise.
"M_Id"	Identity map
"M_Undefined"	Undefined map
"M_Unknown"	Placeholder for an unknown map
"M_ContainedInLowerSet"	Test for containment in a lower set
"M_ContainedInUpperSet"	Test for containment in an upper set
"M_Injection"	Injection into a poset sum
"M_Join"	Join operation
"M_JoinConstant"	Join with a constant value
"M_Meet"	Meet operation
"M_MeetConstant"	Meet with a constant
"M_RepresentPrincipalLowerSet_TotalOrderBounded"	Largest principal lower set in the poset.
"M_RepresentPrincipalUpperSet_TotalOrderBounded"	Largest principal upper set in the poset.
"M_SmashInjection"	Injection into a smash sum
"M_C_Op"	Opposite of a map
"M_C_RefineDomain"	A refinement of the domain of a monotone map
"M_C_WrapUnits"	Wraps a monotone map with units descriptions for domain and codomain.
"M_C_Coproduct"	Coproduct of monotone maps
"M_C_CoproductSmash"	Smash coproduct of two monotone maps
"M_C_DomProdCodSmash"	A monotone map from a product of domains to a smash product of codomains.
"M_C_DomSmashCodProd"	A monotone map from the smash product of domains to the product of codomains.
"M_C_DomUnion"	Domain union of monotone maps
"M_C_Parallel"	Monoidal product of monotone maps
"M_C_ParallelSmash"	Monoidal (smash) product of monotone maps
"M_C_Product"	Product of monotone maps
"M_C_ProductSmash"	Smash product of monotone maps
"M_C_Series"	Series composition of monotone maps
"M_C_Sum"	Sum of monotone maps
"M_C_SumSmash"	Smash sum of monotone maps
"M_AddL"	Addition in the L topology.
"M_AddLConstant"	Add a constant in the L topology.
"M_AddU"	Addition in the U topology.
"M_AddUConstant"	Addition of constant in the U topology.
"M_Ceil0"	Ceiling function relative
"M_DivideLConstant"	Division by a constant (L topology)
"M_DivideUConstant"	Division by a constant (U topology)
"M_Floor0"	Floor function relative
"M_MultiplyL"	Multiplication (L topology)
"M_MultiplyLConstant"	Multiplication by a constant (L topology)
"M_MultiplyU"	Multiplication (U topology)
"M_MultiplyUConstant"	Multiplication by a constant (U topology)
"M_PowerFracL"	Lift to the power of a fraction (L topology)
"M_PowerFracU"	Lift to the power of a fraction (U topology)
"M_RoundDown"	Round down
"M_RoundUp"	Round up
"M_ScaleL"	Scaling in the L topology by a fraction.
"M_ScaleU"	Scaling in the U topology by a fraction.
"M_SubLConstant"	Subtraction of a constant (L topology)
"M_SubUConstant"	Subtraction by a constant (U topology)
"M_C_LiftToSubsets"	Lift of a monotone map to subsets
"M_LiftToLowerSets"	Lifts a monotone map to lower sets
"M_LiftToUpperSets"	Lifts a monotone map to upper sets
"M_BottomIfNotTop"	Maps top to top, and everything else to bottom.
"M_IdentityBelowThreshold"	A monotone map that outputs a constant value if the input is above a threshold.
"M_Threshold1"	Threshold map (r-to-f for DP_FuncNotMoreThan)
"M_Threshold2"	Threshold map (f-to-r for DP_ResNotLessThan)

"M_TopIfNotBottom"	Maps bottom to bottom, and everything else to top.
"M_Lift"	Lifts a value to a tuple with one element.
"M_TakeIndex"	Projection of an element in a poset product.
"M_TakeRange"	Projection of a range of elements in a smash poset product.
"M_Unlift"	Unlifts a one-element tuple to its single element.
"M_C_Leq_X"	Tests $\text{constant} \leq x$
"M_C_Lt_X"	Tests $\text{constant} < x$
"M_X_Leq_C"	Tests $x \leq \text{constant}$
"M_X_Lt_C"	Tests $x < \text{constant}$
"M_Leq"	Tests $x_1 \leq_p x_2$

### 26.3.1. **M\_Constant** - A constant function

Data

**Extends:** `MonotoneMap`(dom, cod, type = "M\_Constant")

Property	Symbol	Type	Description
value	$v$	Value	The constant value of the map.

*Discussed in Section 18.2 (Constant maps)*

### 26.3.2. **M\_Empty** - The unique map from the empty set to another

Data

**Extends:** `MonotoneMap`(dom, cod, type = "M\_Empty")

### 26.3.3. **M\_Explicit** - A map defined pointwise.

Data

**Extends:** `MonotoneMap`(dom, cod, type = "M\_Explicit")

Property	Symbol	Type	Description
options		<code>array[M_Explicit_Option]</code>	A list of pairs $(x, y)$ such that the map sends $x$ to $y$ .

*Discussed in Section 18.9 (Catalog)*

#### **M\_Explicit\_Option**

Data

Property	Symbol	Type	Description
x		any	
y		any	

### 26.3.4. **M\_Id** - Identity map

Data

**Extends:** `MonotoneMap`(dom, cod, type = "M\_Id")

*Discussed in Section 18.1 (Identity map)*

### 26.3.5. **M\_Undefined** - Undefined map

Data

**Extends:** `MonotoneMap`(dom, cod, type = "M\_Undefined")

### 26.3.6. **M\_Unknown** - Placeholder for an unknown map

Data

**Extends:** `MonotoneMap`(dom, cod, type = "M\_Unknown")

### 26.3.7. **M\_ContainedInLowerSet** - Test for containment in a lower set

Data

**Extends:** `MonotoneMap`(dom, cod, type = "M\_ContainedInLowerSet")

Property	Symbol	Type	Description
lower_set	$L$	LowerSet	The lower set
ospace	$P$	Poset	The poset in which the lower set is defined.

*Discussed in Section 18.12.1 (Lower set containment tests)*

This map tests whether the input is contained in the lower set:

$$x \mapsto x \in L \quad (3)$$

### 26.3.8. **M\_ContainedInUpperSet** - Test for containment in an upper set

Data

**Extends:** `MonotoneMap`(dom, cod, type = "M\_ContainedInUpperSet")

Property	Symbol	Type	Description
upper_set	$U$	UpperSet	The upper set
ospace	$P$	Poset	The poset in which the upper set is defined

*Discussed in Section 18.12.2 (Upper set containment tests)*

This map tests whether the input is contained in the upper set:

$$x \mapsto x \in U \quad (4)$$

### 26.3.9. **M\_Injection** - Injection into a poset sum

Data

**Extends:** `MonotoneMap`(dom, cod, type = "M\_Injection")

Property	Symbol	Type	Description
index		integer	Which space to inject into

*Discussed in Section 18.8.2 (Injections)*

This is the injection map into the  $i$ -th poset of a poset sum `P_C_Sum`.

### 26.3.10. **M\_Join** - Join operation

Data

**Extends:** `MonotoneMap`(dom, cod, type = "M\_Join")

Property	Symbol	Type	Description
ospaces		array[Poset]	The posets in which each join is defined.

*Discussed in Section 18.6.1 ( $n$ -ary Join)*

The join operation is defined as follows:

$$x \mapsto \bigvee_i x_i \quad (5)$$

where  $x_i$  is the  $i$ -th component of  $x$  in the poset sum.

### 26.3.11. **M\_JoinConstant** - Join with a constant value

Data

**Extends:** `MonotoneMap`(dom, cod, type = "M\_JoinConstant")

Property	Symbol	Type	Description
value	$c$	Value	The constant value to join with.
ospace	$P$	Poset	The poset in which the join operation is defined.

*Discussed in Section 18.5 (Unary join and meet operations)*

This map performs a join with a constant value:

$$x \mapsto x \vee c \quad (6)$$

where  $c$  is the constant value.

### 26.3.12. **M\_Meet** - Meet operation

Data

**Extends:** `MonotoneMap`(dom, cod, type = "M\_Meet")

Property	Symbol	Type	Description
ospaces		array[Poset]	The posets in which each meet is defined.

*Discussed in Section 18.6.2 ( $n$ -ary Meet)*

The meet operation is defined as follows:

$$x \mapsto \bigwedge_i x_i \quad (7)$$

where  $x_i$  is the  $i$ -th component of  $x$  in the poset sum.

### 26.3.13. **M\_MeetConstant** - Meet with a constant

Data

**Extends:** `MonotoneMap`(dom, cod, type = "M\_MeetConstant")

Property	Symbol	Type	Description
value	$c$	Value	The constant value to meet with.
ospace	$P$	Poset	The poset in which the meet operation is defined.

*Discussed in Section 18.5 (Unary join and meet operations)*

This map performs a meet with a constant value:

$$x \mapsto x \wedge c \quad (8)$$

where  $c$  is the constant value.

### 26.3.14. **M\_RepresentPrincipalLowerSet\_TotalOrderBounded** - Largest principal lower set in the poset.

Data

**Extends:** `MonotoneMap`(dom, cod, type = "M\_RepresentPrincipalLowerSet\_TotalOrderBounded")

This map  $m : \text{dom} \rightarrow \text{cod}$ , with dom, cod subposets of a common ambient posets. takes a point  $x$  in the domain and returns the largest lower set containing its down closure.

$$x \mapsto \arg \max_{y \leq x} y \quad (9)$$

The compilation process ensures that this map is constructed only when this is always well defined.

Example: dom =  $\mathbb{R}$  and cod =  $\mathbb{N}$

### 26.3.15. **M\_RepresentPrincipalUpperSet\_TotalOrderBounded** - Largest principal upper set in the poset.

Data

**Extends:** `MonotoneMap`(dom, cod, type = "M\_RepresentPrincipalUpperSet\_TotalOrderBounded")

This map  $m : \text{dom} \rightarrow \text{cod}$ , with dom, cod subposets of a common ambient poset takes a point  $x$  in the domain and returns the largest upper set containing its closure.

$$x \mapsto \arg \min_{y \geq x} y$$

The compilation process ensures that this map is constructed only when this is always well defined.

Example: dom =  $\mathbb{R}$  and cod =  $\mathbb{N}$

### 26.3.16. **M\_SmashInjection** - Injection into a smash sum

Data

**Extends:** `MonotoneMap`(dom, cod, type = "M\_SmashInjection")

Property	Symbol	Type	Description
index		integer	Which space to inject into

*Discussed in Section 18.8.2 (Injections)*

This is the injection map into the  $i$ -th poset of a smash poset sum `P_C_SumSmash`.



### 26.3.17. **M\_C\_Op** - Opposite of a map

Data

**Extends:** `MonotoneMap`(dom, cod, type = "M\_C\_Op")

Property	Symbol	Type	Description
<code>m</code>	$m$	<code>MonotoneMap</code>	The original map

*Discussed in Section 19.1.1 (Opposite of a map)*

### 26.3.18. **M\_C\_RefineDomain** - A refinement of the domain of a monotone map

Data

**Extends:** `MonotoneMap`(dom, cod, type = "M\_C\_RefineDomain")

Property	Symbol	Type	Description
<code>m</code>	$m$	<code>MonotoneMap</code>	The original map

### 26.3.19. **M\_C\_WrapUnits** - Wraps a monotone map with units descriptions for domain and codomain.

Data

**Extends:** `MonotoneMap`(dom, cod, type = "M\_C\_WrapUnits")

Property	Symbol	Type	Description
<code>m</code>	$m$	<code>MonotoneMap</code>	The original map
<code>dom_units</code>		<code>Unit</code>	Units for the domain of the monotone map.
<code>cod_units</code>		<code>Unit</code>	Units for the codomain of the monotone map.

### 26.3.20. **M\_C\_Coproduct** - Coproduct of monotone maps

Data

**Extends:** `MonotoneMap`(dom, cod, type = "M\_C\_Coproduct")

Property	Symbol	Type	Description
<code>maps</code>	$m_i$	<code>array[MonotoneMap]</code>	A list of monotone maps that are composed together.
<code>labels</code>		<code>array[string]?</code>	A list of labels for the monotone maps.

*Discussed in Section 19.2.5 (Coproduct of maps)*

The coproduct of two or more monotone maps is a monotone map that selects one or the other map depending on the input.

If  $f : A \rightarrow B$  and  $g : C \rightarrow B$ , then

$$\mathbf{M\_C\_Coproduct}(f, g) : \mathbf{P\_C\_Sum}(A, C) \rightarrow B \quad (10)$$

### 26.3.21. [M\\_C\\_CoproductSmash](#) - Smash coproduct of two monotone maps

Data

**Extends:** [MonotoneMap](#)(dom, cod, type = "M\_C\_CoproductSmash")

Property	Symbol	Type	Description
<a href="#">maps</a>	$m_i$	array[ <a href="#">MonotoneMap</a> ]	A list of monotone maps that are composed together.
labels		array[string]?	A list of labels for the monotone maps.

*Discussed in Section 19.2.5 (Coproduct of maps)*

The smash coproduct of two or more monotone maps is a monotone map that selects one or the other map depending on the input.

If  $f : A \rightarrow B$  and  $g : C \rightarrow B$ , then

$$\text{M\_C\_CoproductSmash}(f, g) : \text{P\_C\_SumSmash}(A, C) \rightarrow B \quad (11)$$

### 26.3.22. [M\\_C\\_DomProdCodSmash](#) - A monotone map from a product of domains to a smash product of codomains.

Data

**Extends:** [MonotoneMap](#)(dom, cod, type = "M\_C\_DomProdCodSmash")

Property	Symbol	Type	Description
<a href="#">maps</a>	$m_i$	array[ <a href="#">MonotoneMap</a> ]	A list of monotone maps that are composed together.
labels		array[string]?	A list of labels for the monotone maps.

*Discussed in Section 19.2.1 (Parallel composition)*

For two monotone maps  $f : A \rightarrow B$  and  $g : C \rightarrow D$ , we can define the monotone map

$$\text{M\_C\_DomProdCodSmash}(f, g) : \text{P\_C\_Product}(A, C) \rightarrow \text{P\_C\_ProductSmash}(B, D) \quad (12)$$

in the obvious way.

### 26.3.23. [M\\_C\\_DomSmashCodProd](#) - A monotone map from the smash product of domains to the product of codomains.

Data

**Extends:** [MonotoneMap](#)(dom, cod, type = "M\_C\_DomSmashCodProd")

Property	Symbol	Type	Description
<a href="#">maps</a>	$m_i$	array[ <a href="#">MonotoneMap</a> ]	A list of monotone maps that are composed together.
labels		array[string]?	A list of labels for the monotone maps.

*Discussed in Section 19.2.1 (Parallel composition)*

For two monotone maps  $f : A \rightarrow B$  and  $g : C \rightarrow D$ , we can define the monotone map

$$\text{M\_C\_DomSmashCodProd}(f, g) : \text{P\_C\_ProductSmash}(A, C) \rightarrow \text{P\_C\_Product}(B, D) \quad (13)$$

in the obvious way.

#### 26.3.24. **M\_C\_DomUnion** - Domain union of monotone maps

Data

**Extends:** `MonotoneMap`(dom, cod, type = "M\_C\_DomUnion")

Property	Symbol	Type	Description
<code>maps</code>	$m_i$	<code>array[MonotoneMap]</code>	A list of monotone maps that are composed together.
<code>labels</code>		<code>array[string]?</code>	A list of labels for the monotone maps.

*Discussed in Section 19.2.6 (Domain union)*

Given two monotone maps  $f : A \rightarrow B$  and  $g : C \rightarrow B$ , the domain union of these maps is a monotone map that combines the domains of both maps:

$$\mathbf{M\_C\_DomUnion}(f, g) : \mathbf{P\_F\_C\_Union}(A, C) \rightarrow B \quad (14)$$

The value is defined as follows:

$$\mathbf{M\_C\_DomUnion}(f, g) : x \mapsto \begin{cases} f(x) & \text{if } x \in A \\ g(x) & \text{if } x \in C \end{cases} \quad (15)$$

Note that the order of the maps does matter. We use the first map whose domain contains the input.

#### 26.3.25. **M\_C\_Parallel** - Monoidal product of monotone maps

Data

**Extends:** `MonotoneMap`(dom, cod, type = "M\_C\_Parallel")

Property	Symbol	Type	Description
<code>maps</code>	$m_i$	<code>array[MonotoneMap]</code>	A list of monotone maps that are composed together.
<code>labels</code>		<code>array[string]?</code>	A list of labels for the monotone maps.

*Discussed in Section 19.2.1 (Parallel composition)*

If  $f : A \rightarrow B$  and  $g : C \rightarrow D$ , then

$$\mathbf{M\_C\_Parallel}(f, g) : \mathbf{P\_C\_Product}(A, C) \rightarrow \mathbf{P\_C\_Product}(B, D) \quad (16)$$

#### 26.3.26. **M\_C\_ParallelSmash** - Monoidal (smash) product of monotone maps

Data

**Extends:** `MonotoneMap`(dom, cod, type = "M\_C\_ParallelSmash")

Property	Symbol	Type	Description
<code>maps</code>	$m_i$	<code>array[MonotoneMap]</code>	A list of monotone maps that are composed together.
<code>labels</code>		<code>array[string]?</code>	A list of labels for the monotone maps.

*Discussed in Section 19.2.1 (Parallel composition)*

If  $f : A \rightarrow B$  and  $g : C \rightarrow D$ , then

$$\mathbf{M\_C\_ParallelSmash}(f, g) : \mathbf{P\_C\_ProductSmash}(A, C) \rightarrow \mathbf{P\_C\_ProductSmash}(B, D) \quad (17)$$

#### 26.3.27. **M\_C\_Product** - Product of monotone maps

Data

**Extends:** `MonotoneMap`(dom, cod, type = "M\_C\_Product")

Property	Symbol	Type	Description
<code>maps</code>	$m_i$	<code>array[MonotoneMap]</code>	A list of monotone maps that are composed together.

Data

labels		array[string]?	A list of labels for the monotone maps.
--------	--	----------------	---

*Discussed in Section 19.2.3 (Product of maps)*

The product of two or more monotone maps with the same domain is a monotone map that combines the outputs of both maps in a product.

If  $f : A \rightarrow B$  and  $g : A \rightarrow C$ , then

$$\mathbf{M\_C\_Product}(f, g) : A \rightarrow \mathbf{P\_C\_Product}(B, D) \quad (18)$$

### 26.3.28. **M\_C\_ProductSmash** - Smash product of monotone maps

Data

**Extends:** **MonotoneMap**(dom, cod, type = "M\_C\_ProductSmash")

Property	Symbol	Type	Description
maps	$m_i$	array[ <b>MonotoneMap</b> ]	A list of monotone maps that are composed together.
labels		array[string]?	A list of labels for the monotone maps.

*Discussed in Section 19.2.3 (Product of maps)*

The product of two or more monotone maps with the same domain is a monotone map that combines the outputs of both maps in a smash product.

If  $f : A \rightarrow B$  and  $g : A \rightarrow C$ , then

$$\mathbf{M\_C\_ProductSmash}(f, g) : A \rightarrow \mathbf{P\_C\_ProductSmash}(B, D) \quad (19)$$

### 26.3.29. **M\_C\_Series** - Series composition of monotone maps

Data

**Extends:** **MonotoneMap**(dom, cod, type = "M\_C\_Series")

Property	Symbol	Type	Description
maps	$m_i$	array[ <b>MonotoneMap</b> ]	A list of monotone maps that are composed together.
labels		array[string]?	A list of labels for the monotone maps.

*Discussed in Section 19.2.2 (Series composition)*

### 26.3.30. **M\_C\_Sum** - Sum of monotone maps

Data

**Extends:** **MonotoneMap**(dom, cod, type = "M\_C\_Sum")

Property	Symbol	Type	Description
maps	$m_i$	array[ <b>MonotoneMap</b> ]	A list of monotone maps that are composed together.
labels		array[string]?	A list of labels for the monotone maps.

*Discussed in Section 19.2.4 (Sum of maps)*

The sum of two or more monotone maps is a monotone map that combines the outputs of both maps.

If  $f : A \rightarrow B$  and  $g : C \rightarrow D$ , then

$$\mathbf{M\_C\_Sum}(f, g) : \mathbf{P\_C\_Sum}(A, C) \rightarrow \mathbf{P\_C\_Sum}(B, D) \quad (20)$$

### 26.3.31. [M\\_C\\_SumSmash](#) - Smash sum of monotone maps

Data

**Extends:** [MonotoneMap](#)(dom, cod, type = "M\_C\_SumSmash")

Property	Symbol	Type	Description
<a href="#">maps</a>	$m_i$	array[ <a href="#">MonotoneMap</a> ]	A list of monotone maps that are composed together.
labels		array[string]?	A list of labels for the monotone maps.

*Discussed in Section 19.2.4 (Sum of maps)*

The smash sum of two or more monotone maps is a monotone map that combines the outputs of both maps.

If  $f : A \rightarrow B$  and  $g : C \rightarrow D$ , then

$$\text{M\_C\_SumSmash}(f, g) : \text{P\_C\_SumSmash}(A, C) \rightarrow \text{P\_C\_SumSmash}(B, D) \quad (21)$$

### 26.3.32. [M\\_AddL](#) - Addition in the L topology.

Data

**Extends:** [MonotoneMap](#)(dom, cod, type = "M\_AddL")

Property	Symbol	Type	Description
ospace	$P_i$	array[ <a href="#">Poset</a> ]	The sequence of posets in which to perform the operation.

This map takes a sequence of elements from posets and returns their sum in the L topology.

If the domain is a product of length  $n$ , there are  $n - 1$  in opspaces.

For example, if the domain is a product of 4 posets, the sum  $a + b + c + d$  is computed as follows:

$$\begin{aligned} t_1 &= a +_{\text{ospace}_0} b \\ t_2 &= t_1 +_{\text{ospace}_1} c \\ t_3 &= t_2 +_{\text{ospace}_2} d \end{aligned}$$

### 26.3.33. [M\\_AddLConstant](#) - Add a constant in the L topology.

Data

**Extends:** [MonotoneMap](#)(dom, cod, type = "M\_AddLConstant")

Property	Symbol	Type	Description
ospace	$P$	<a href="#">Poset</a>	The poset in which we do the operation.
value	$c$	Value	The constant to add to the input.

The map  $x \mapsto x + c$ .

### 26.3.34. [M\\_AddU](#) - Addition in the U topology.

Data

**Extends:** [MonotoneMap](#)(dom, cod, type = "M\_AddU")

Property	Symbol	Type	Description
ospace	$P_i$	array[ <a href="#">Poset</a> ]	The sequence of posets in which to perform the operation.

See [M\\_AddL](#) for the meaning of opspaces.

### 26.3.35. [M\\_AddUConstant](#) - Addition of constant in the U topology.

Data

**Extends:** [MonotoneMap](#)(dom, cod, type = "M\_AddUConstant")

Property	Symbol	Type	Description
ospace value	$P$ $c$	Poset Value	The poset in which we do the operation. The constant to add.

The map  $x \mapsto x + c$ .

### 26.3.36. [M\\_Ceil0](#) - Ceiling function relative

Data

**Extends:** [MonotoneMap](#)(dom, cod, type = "M\_Ceil0")

Property	Symbol	Type	Description
ospace	$P$	Poset	The poset in which we do the operation.

This is a relative of the ceiling function defined only on  $\mathbb{R}_{\geq 0}$  and defined as follows:

$$x \mapsto \begin{cases} 0 & \text{if } x = 0 \\ \infty & \text{if } x = \infty \\ \text{floor}(x + 1) & \text{otherwise} \end{cases} \quad (22)$$

See [M\\_RoundUp](#) for implementing the regular ceiling function.

### 26.3.37. [M\\_DivideLConstant](#) - Division by a constant (L topology)

Data

**Extends:** [MonotoneMap](#)(dom, cod, type = "M\_DivideLConstant")

Property	Symbol	Type	Description
ospace value	$P$ $c$	Poset Value	The poset in which we do the operation. The constant to divide by.

*Discussed in Section 18.4.3 (Division)*

The map  $x \mapsto x/c$ .

### 26.3.38. [M\\_DivideUConstant](#) - Division by a constant (U topology)

Data

**Extends:** [MonotoneMap](#)(dom, cod, type = "M\_DivideUConstant")

Property	Symbol	Type	Description
ospace value	$P$ $c$	Poset Value	The poset in which we do the operation. The constant to divide by.

*Discussed in Section 18.4.3 (Division)*

The map  $x \mapsto x/c$ .

### 26.3.39. [M\\_Floor0](#) - Floor function relative

Data

**Extends:** [MonotoneMap](#)(dom, cod, type = "M\_Floor0")

Property	Symbol	Type	Description
ospace	$P$	Poset	The poset in which we do the operation.

This is a relative of the floor function defined only on  $\mathbb{R}_{\geq 0}$  and defined as follows:

$$x \mapsto \begin{cases} 0 & \text{if } x = 0 \\ \infty & \text{if } x = \infty \\ \text{ceil}(x - 1) & \text{otherwise} \end{cases} \quad (23)$$

See [M\\_RoundDown](#) for implementing the regular floor function.

### 26.3.40. [M\\_MultiplyL](#) - Multiplication (L topology)

Data

**Extends:** [MonotoneMap](#)(dom, cod, type = "M\_MultiplyL")

Property	Symbol	Type	Description
ospaces	$P_i$	array[Poset]	The sequence of posets in which to perform the operation.

See [M\\_AddL](#) for the meaning of the optional parameters.

### 26.3.41. [M\\_MultiplyLConstant](#) - Multiplication by a constant (L topology)

Data

**Extends:** [MonotoneMap](#)(dom, cod, type = "M\_MultiplyLConstant")

Property	Symbol	Type	Description
ospace	$P$	Poset	The poset in which we do the operation.
value	$c$	Value	The constant to multiply by.

The map  $x \mapsto x \cdot c$ .

### 26.3.42. [M\\_MultiplyU](#) - Multiplication (U topology)

Data

**Extends:** [MonotoneMap](#)(dom, cod, type = "M\_MultiplyU")

Property	Symbol	Type	Description
ospaces	$P_i$	array[Poset]	The sequence of posets in which to perform the operation.

*Discussed in Section 18.4.2 (Multiplication)*

See [M\\_AddL](#) for the meaning of ospaces.

### 26.3.43. [M\\_MultiplyUConstant](#) - Multiplication by a constant (U topology)

Data

**Extends:** [MonotoneMap](#)(dom, cod, type = "M\_MultiplyUConstant")

Property	Symbol	Type	Description
ospace	$P$	Poset	The poset in which we do the operation.
value	$c$	Value	The constant to multiply by.

The map  $x \mapsto x \cdot c$ .

#### 26.3.44. **M\_PowerFracL** - Lift to the power of a fraction (L topology)

Data	<b>Extends:</b> <code>MonotoneMap</code> (dom, cod, type = "M_PowerFracL")			
	Property	Symbol	Type	Description
	ospace	$P$	Poset	The poset in which we do the operation.
	num	$N$	string	The numerator of the fraction.
	den	$D$	string	The denominator of the fraction.

The map  $x \mapsto x^{N/D}$ .

#### 26.3.45. **M\_PowerFracU** - Lift to the power of a fraction (U topology)

Data	<b>Extends:</b> <code>MonotoneMap</code> (dom, cod, type = "M_PowerFracU")			
	Property	Symbol	Type	Description
	ospace	$P$	Poset	The poset in which we do the operation.
	num	$N$	string	The numerator of the fraction.
	den	$D$	string	The denominator of the fraction.

The map  $x \mapsto x^{N/D}$ .

#### 26.3.46. **M\_RoundDown** - Round down

Data	<b>Extends:</b> <code>MonotoneMap</code> (dom, cod, type = "M_RoundDown")			
	Property	Symbol	Type	Description
	ospace	$P$	Poset	The poset in which we do the operation.
	offset	$o$	any	The offset to subtract before rounding down.
	step	$s$	string	The step to round down to.

*Discussed in Section 18.3.1 (Generalized rounding)*

#### 26.3.47. **M\_RoundUp** - Round up

Data	<b>Extends:</b> <code>MonotoneMap</code> (dom, cod, type = "M_RoundUp")			
	Property	Symbol	Type	Description
	ospace	$P$	Poset	The poset in which we do the operation.
	offset	$o$	any	The offset to subtract before rounding up.
	step	$s$	string	The step to round up to.

*Discussed in Section 18.3.1 (Generalized rounding)*

#### 26.3.48. **M\_ScaleL** - Scaling in the L topology by a fraction.

Data	<b>Extends:</b> <code>MonotoneMap</code> (dom, cod, type = "M_ScaleL")			
	Property	Symbol	Type	Description
	ospace	$P$	Poset	The poset in which we do the operation.
	num	$N$	string	The numerator of the fraction.
	den	$D$	string	The denominator of the fraction.

The map  $x \mapsto x \cdot (N/D)$ .



### 26.3.49. [M\\_ScaleU](#) - Scaling in the U topology by a fraction.

Data

**Extends:** [MonotoneMap](#)(dom, cod, type = "M\_ScaleU")

Property	Symbol	Type	Description
ospace	$P$	Poset	The poset in which we do the operation.
num	$N$	string	The numerator of the fraction.
den	$D$	string	The denominator of the fraction.

The map  $x \mapsto x \cdot_p (N/D)$ .

### 26.3.50. [M\\_SubLConstant](#) - Subtraction of a constant (L topology)

Data

**Extends:** [MonotoneMap](#)(dom, cod, type = "M\_SubLConstant")

Property	Symbol	Type	Description
ospace	$P$	Poset	The poset in which we do the operation.
value	$c$	Value	The constant to subtract.

The map  $x \mapsto x - c$ .

### 26.3.51. [M\\_SubUConstant](#) - Subtraction by a constant (U topology)

Data

**Extends:** [MonotoneMap](#)(dom, cod, type = "M\_SubUConstant")

Property	Symbol	Type	Description
ospace	$P$	Poset	The poset in which we do the operation.
value	$c$	Value	The constant to subtract.

The map  $x \mapsto x - c$ .

### 26.3.52. [M\\_C\\_LiftToSubsets](#) - Lift of a monotone map to subsets

Data

**Extends:** [MonotoneMap](#)(dom, cod, type = "M\_C\_LiftToSubsets")

Property	Symbol	Type	Description
<a href="#">m</a>		<a href="#">MonotoneMap</a>	The monotone map that is lifted.

*Discussed in Section 18.7 (Lifts to subsets)*

### 26.3.53. [M\\_LiftToLowerSets](#) - Lifts a monotone map to lower sets

Data

**Extends:** [MonotoneMap](#)(dom, cod, type = "M\_LiftToLowerSets")

Property	Symbol	Type	Description
<a href="#">m</a>		<a href="#">MonotoneMap</a>	The monotone map that is lifted.

*Discussed in Section 18.7 (Lifts to subsets)*

#### 26.3.54. **M\_LiftToUpperSets** - Lifts a monotone map to upper sets

Data

**Extends:** `MonotoneMap`(dom, cod, type = "M\_LiftToUpperSets")

Property	Symbol	Type	Description
<code>m</code>		<code>MonotoneMap</code>	The monotone map that is lifted.

*Discussed in Section 18.7 (Lifts to subsets)*

#### 26.3.55. **M\_BottomIfNotTop** - Maps top to top, and everything else to bottom.

Data

**Extends:** `MonotoneMap`(dom, cod, type = "M\_BottomIfNotTop")

*Discussed in Section 18.10 (Threshold maps)*

$$x \mapsto \begin{cases} \top_C & \text{if } x = \top_D \\ \perp_C & \text{otherwise} \end{cases} \quad (24)$$

#### 26.3.56. **M\_IdentityBelowThreshold** - A monotone map that outputs a constant value if the input is above a threshold.

Data

**Extends:** `MonotoneMap`(dom, cod, type = "M\_IdentityBelowThreshold")

Property	Symbol	Type	Description
<code>value</code>	$V$	Value	Value returned by the map if the input is above the threshold. This value must be greater than or equal to the threshold.
<code>threshold</code>	$T$	Value	Threshold value.

*Discussed in Section 18.10 (Threshold maps)*

$$x \mapsto \begin{cases} V & \text{if } T \leq x \\ x & \text{otherwise} \end{cases} \quad (25)$$

where  $T \leq V$ .

#### 26.3.57. **M\_Threshold1** - Threshold map (r-to-f for `DP_FuncNotMoreThan`)

Data

**Extends:** `MonotoneMap`(dom, cod, type = "M\_Threshold1")

Property	Symbol	Type	Description
<code>value</code>	$v$	Value	A constant value

*Discussed in Section 18.10 (Threshold maps)*

This map returns a constant value:

$$x \mapsto \begin{cases} x & \text{if } x \leq v \\ v & \text{otherwise} \end{cases} \quad (26)$$

This map is the r-to-f map for `DP_FuncNotMoreThan`

### 26.3.58. **M\_Threshold2** - Threshold map (f-to-r for DP\_ResNotLessThan)

Data

**Extends:** `MonotoneMap`(dom, cod, type = "M\_Threshold2")

Property	Symbol	Type	Description
value	$v$	Value	A constant value

*Discussed in Section 18.10 (Threshold maps)*

This map returns a constant value:

$$x \mapsto \begin{cases} x & \text{if } v \leq x \\ v & \text{otherwise} \end{cases} \quad (27)$$

This map arises as the f-to-r map of `DP_ResNotLessThan`.

### 26.3.59. **M\_TopIfNotBottom** - Maps bottom to bottom, and everything else to top.

Data

**Extends:** `MonotoneMap`(dom, cod, type = "M\_TopIfNotBottom")

*Discussed in Section 18.10 (Threshold maps)*

$$x \mapsto \begin{cases} \perp_C & \text{if } x = \perp_D \\ \top_C & \text{otherwise} \end{cases} \quad (28)$$

### 26.3.60. **M\_Lift** - Lifts a value to a tuple with one element.

Data

**Extends:** `MonotoneMap`(dom, cod, type = "M\_Lift")

*Discussed in Section 18.8 (Plumbing)*

The map is  $x \mapsto \langle x \rangle$

This is the inverse of `M_Unlift`.

### 26.3.61. **M\_TakeIndex** - Projection of an element in a poset product.

Data

**Extends:** `MonotoneMap`(dom, cod, type = "M\_TakeIndex")

Property	Symbol	Type	Description
projection		Projection	Describes the projection

*Discussed in Section 18.8.1 (Slicing)*

The map  $x \mapsto x_i$  where  $i$  is the index of the projection.

**Projection - Projection from a product.**

Data

Property	Symbol	Type	Description
type		string	Type marker <i>Must be equal to "Projection"</i>
index		integer	The index of the element to project.

Data	ntot		integer		The total number of elements in the product.
------	------	--	---------	--	--

26.3.62. **M\_TakeRange** - Projection of a range of elements in a smash poset product.

Data	Extends: MonotoneMap(dom, cod, type = "M_TakeRange")				
Property	Symbol	Type	Description		
range		Range	Describes the range of indices to take.		

Discussed in Section 18.8.1 (Slicing)

The map  $x \mapsto x_{i..j}$  where  $i$  and  $j$  are the bounds of the range.

26.3.63. **M\_Unlift** - Unlifts a one-element tuple to its single element.

Data	Extends: MonotoneMap(dom, cod, type = "M_Unlift")				
------	---	--	--	--	--

Discussed in Section 18.8 (Plumbing)

The map is  $\langle x \rangle \mapsto x$   
This is the inverse of [M\\_Lift](#).

26.3.64. **M\_C\_Leq\_X** - Tests constant  $\leq x$

Data	Extends: MonotoneMap(dom, cod, type = "M_C_Leq_X")				
Property	Symbol	Type	Description		
ospace value	$P$ $v$	Poset Value	Poset in which the comparison is performed. Comparison value.		

Discussed in Section 18.11.1 (constant  $\leq x$ )

The map  $x \mapsto v \leq_p x$ .

26.3.65. **M\_C\_Lt\_X** - Tests constant  $< x$

Data	Extends: MonotoneMap(dom, cod, type = "M_C_Lt_X")				
Property	Symbol	Type	Description		
ospace value	$P$ $v$	Poset Value	Poset in which the comparison is performed. Comparison value.		

Discussed in Section 18.11.2 (constant  $< x$ )

The map  $x \mapsto v <_p x$ .

### 26.3.66. $\mathbf{M\_X\_Leq\_C}$ - Tests $x \leq \text{constant}$

Data

**Extends:** `MonotoneMap`(dom, cod, type = "M\_X\_Leq\_C")

Property	Symbol	Type	Description
ospace value	$P$ $v$	Poset Value	Poset in which the comparison is performed. Comparison value.

*Discussed in Section 18.11.3 ( $x \leq \text{constant}$ )*

The map  $x \mapsto x \leq_P v$ .

### 26.3.67. $\mathbf{M\_X\_Lt\_C}$ - Tests $x < \text{constant}$

Data

**Extends:** `MonotoneMap`(dom, cod, type = "M\_X\_Lt\_C")

Property	Symbol	Type	Description
ospace value	$P$ $v$	Poset Value	Poset in which the comparison is performed. Comparison value.

*Discussed in Section 18.11.4 ( $x < \text{constant}$ )*

The map  $x \mapsto x <_P v$ .

### 26.3.68. $\mathbf{M\_Leq}$ - Tests $x_1 \leq_P x_2$

Data

**Extends:** `MonotoneMap`(dom, cod, type = "M\_Leq")

Property	Symbol	Type	Description
ospace	$P$	Poset	Poset in which the comparison is performed.

*Discussed in Section 18.13 (Order as a function)*

The map  $(x_1, x_2) \mapsto x_1 \leq_P x_2$ .

## 26.4. L1Map - Map to lower sets of functionalities.

Data

**Extends:** Root(version, description, hash, kind = "L1Map")

Property	Symbol	Type	Description
kdom	kdom	Poset	Kleisli domain of the map
kcod	kcod	Poset	Kleisli co-domain of the map
type		string	Discriminator variable to distinguish subtypes.

Discussed in Section 11.2 (**PosL** and **PosU**)

Subtypes based on the value for type

"L1_Constant"	Constant map
"L1_Entire"	Returns the entire poset
"L1_Explicit"	Map defined pointwise
"L1_Identity"	Lift of the identity map
"L1_Unknown"	Placeholder for an unknown map.
"L1_Catalog"	Map induced by a catalog of options.
"L1_IntersectionOfPrinLowerSets"	Intersection of principal lower sets.
"L1_RepresentPrincipalLowerSet"	Represent a principal lower set
"L1_UnionOfPrinLowerSets"	Union of principal lower sets.
"L1_C_CodSum"	Co-domain sum combination
"L1_C_CodSumSmash"	Co-domain (smash) sum combination
"L1_C_DomUnion"	Domain union
"L1_C_Parallel"	Monoidal product
"L1_C_ProdIntersection"	From product to intersection
"L1_C_Product"	Product
"L1_C_Series"	Series composition
"L1_C_Intersection"	Intersection
"L1_C_Union"	Union
"L1_C_RefineDomain"	Refines the domain of a monotone map.
"L1_C_Trace"	Trace
"L1_C_WrapUnits"	Decorates a map with units.
"L1_InvMul_Opt"	Finite-resolution optimistic approximation of the inverse of a multiplication map.
"L1_InvMul_Pes"	Finite-resolution pessimistic approximation of the inverse of an addition map.
"L1_InvSum_Opt"	Finite-resolution optimistic approximation of the inverse of a multiplication map.
"L1_InvSum_Pes"	Finite-resolution pessimistic approximation of the inverse of an addition map.
"L1_FromFilter"	Filters based on a monotone map.
"L1_L_Linv"	Lower inverse of a monotone map
"L1_Lift"	Lifts a monotone map
"L1_TopAlternating"	Lower inverse for the meet map

### 26.4.1. L1\_Constant - Constant map

Data

**Extends:** L1Map(kdom, kcod, type = "L1\_Constant")

Property	Symbol	Type	Description
value	$L$	LowerSet	The constant value of the map, which is a lower set $L \subseteq \text{kcod}$ .

This is a constant map, which maps every element of the domain to the same value  $L$ .

#### 26.4.2. **L1\_Entire** - Returns the entire poset

Data

**Extends:** `L1Map(kdom, kcod, type = "L1_Entire")`

This is the map defined as

$$x \mapsto \text{kcod}$$

which maps every element of the domain to the entire codomain poset `kcod`.

This is useful when the codomain doesn't have a compact representation in terms of antichains.

#### 26.4.3. **L1\_Explicit** - Map defined pointwise

Data

**Extends:** `L1Map(kdom, kcod, type = "L1_Explicit")`

Property	Symbol	Type	Description
options		array[L1_Explicit_Option]	Pairs of input-output

This is a map defined pointwise, where each option specifies a point in the domain and its corresponding value in the codomain.

##### **L1\_Explicit\_Option**

Data

Property	Symbol	Type	Description
x		any	A point in the domain of the map.
y		LowerSet	The lower set corresponding to the point x in the domain.

#### 26.4.4. **L1\_Identity** - Lift of the identity map

Data

**Extends:** `L1Map(kdom, kcod, type = "L1_Identity")`

*Discussed in Section 20.1 (Identity morphisms)*

The identity map:

$$x \mapsto \{x\}$$

#### 26.4.5. **L1\_Unknown** - Placeholder for an unknown map.

Data

**Extends:** `L1Map(kdom, kcod, type = "L1_Unknown")`

This is a placeholder for a map whose type is not known.

#### 26.4.6. L1\_Catalog - Map induced by a catalog of options.

Data

**Extends:** `L1Map(kdom, kcod, type = "L1_Catalog")`

Property	Symbol	Type	Description
options	$\langle f \rangle_k, r_k^*$	array[L1_Catalog_Options]	The options in the catalog

*Discussed in Section 20.3 (Catalog maps)*

This map is defined by a catalog of options  $\llbracket \langle f_k, r_k^* \rangle \rrbracket$ :

$$x \mapsto \bigcup_k \{ \downarrow f_k \mid x \leq r_k^* \} \quad (29)$$

#### L1\_Catalog\_Options - An option in the catalog

Data

Property	Symbol	Type	Description
f	$f$	any	A functionality
r	$r^*$	any	A resource

An option in the catalog, where  $f$  is a point in the domain and  $r$  is a resource.

#### 26.4.7. L1\_IntersectionOfPrinLowerSets - Intersection of principal lower sets.

Data

**Extends:** `L1Map(kdom, kcod, type = "L1_IntersectionOfPrinLowerSets")`

*Discussed in Section 20.4 (Union and intersection of principal lower sets)*

The domain `kdom` must be a product poset.

This map is defined as follows:

$$\langle x_i \rangle_i \mapsto \bigcap_i \downarrow x_i \quad (30)$$

#### 26.4.8. L1\_RepresentPrincipalLowerSet - Represent a principal lower set

Data

**Extends:** `L1Map(kdom, kcod, type = "L1_RepresentPrincipalLowerSet")`

*Discussed in Section 20.5 (Representing principal lower and upper sets)*

This map is defined when `kdom` and `kcod` are subsets of a common ambient poset  $P$ .

#### 26.4.9. L1\_UnionOfPrinLowerSets - Union of principal lower sets.

Data

**Extends:** `L1Map(kdom, kcod, type = "L1_UnionOfPrinLowerSets")`

*Discussed in Section 20.4 (Union and intersection of principal lower sets)*

The domain `kdom` must be a product poset.

This map is defined as follows:

$$\langle x_i \rangle_i \mapsto \bigcup_i \downarrow x_i \quad (31)$$



#### 26.4.10. L1\_C\_CodSum - Co-domain sum combination

Data

**Extends:** `L1Map(kdom, kcod, type = "L1_C_CodSum")`

Property	Symbol	Type	Description
<code>ms</code> labels	$\ell_k$	<code>array[L1Map]</code> <code>array[string]?</code>	Maps to be composed. A list of labels for the maps

*Discussed in Section 20.10 (Codomain Sum)*

Given two maps  $f : A \rightarrow B$  and  $g : A \rightarrow C$ , the codomain sum of these maps is a map that combines the codomains of both maps:

$$\text{L1\_C\_CodSum}(f, g) : A \rightarrow \text{P\_C\_Sum}(B, C) \quad (32)$$

It is defined as follows:

$$x \mapsto \{\text{in}_1(b) \mid b \in f(x)\} \cup \{\text{in}_2(c) \mid c \in g(x)\} \quad (33)$$

#### 26.4.11. L1\_C\_CodSumSmash - Co-domain (smash) sum combination

Data

**Extends:** `L1Map(kdom, kcod, type = "L1_C_CodSumSmash")`

Property	Symbol	Type	Description
<code>ms</code> labels	$\ell_k$	<code>array[L1Map]</code> <code>array[string]?</code>	Maps to be composed. A list of labels for the maps

Given two maps  $f : A \rightarrow B$  and  $g : A \rightarrow C$ , the codomain sum of these maps is a map that combines the codomains of both maps:

$$\text{L1\_C\_CodSumSmash}(f, g) : A \rightarrow \text{P\_C\_SumSmash}(B, C) \quad (34)$$

It is defined as follows:

$$x \mapsto \{\text{in}_1(b) \mid b \in f(x)\} \cup \{\text{in}_2(c) \mid c \in g(x)\} \quad (35)$$

#### 26.4.12. L1\_C\_DomUnion - Domain union

Data

**Extends:** `L1Map(kdom, kcod, type = "L1_C_DomUnion")`

Property	Symbol	Type	Description
<code>ms</code> labels	$\ell_k$	<code>array[L1Map]</code> <code>array[string]?</code>	Maps to be composed. A list of labels for the maps

This is the equivalent of [M\\_C\\_DomUnion](#).

Given two maps  $f : A \rightarrow B$  and  $g : C \rightarrow B$ , the domain union of these maps is a map that combines the domains of both maps:

$$\text{L1\_C\_DomUnion}(f, g) : \text{P\_F\_Union}(A, C) \rightarrow B \quad (36)$$

The value is defined as follows:

$$\text{L1\_C\_DomUnion}(f, g) : x \mapsto \begin{cases} f(x) & \text{if } x \in A \\ g(x) & \text{if } x \in C \end{cases} \quad (37)$$

Note that the order of the maps does matter. We use the first map whose domain contains the input.

### 26.4.13. L1\_C\_Parallel - Monoidal product

Data

**Extends:** L1Map(kdom, kcod, type = "L1\_C\_Parallel")

Property	Symbol	Type	Description
ms labels	$\ell_k$	array[L1Map] array[string]?	Maps to be composed. A list of labels for the maps

*Discussed in Section 20.8 (Parallel composition)*

For a family of maps  $m_i : X_i \rightarrow Y_i$ , the monoidal product is defined as:

$$\text{L1\_C\_Parallel}(\{m_i\}) : \text{P\_C\_Product}(\{X_i\}) \rightarrow \text{P\_C\_Product}(\{Y_i\})$$

### 26.4.14. L1\_C\_ProdIntersection - From product to intersection

Data

**Extends:** L1Map(kdom, kcod, type = "L1\_C\_ProdIntersection")

Property	Symbol	Type	Description
ms labels	$\ell_k$	array[L1Map] array[string]?	Maps to be composed. A list of labels for the maps

The domain **kdom** should be a product.

The map is defined as follows:

$$\langle x_i \rangle_i \mapsto \bigcap_i f_i(x_i) \quad (38)$$

### 26.4.15. L1\_C\_Product - Product

Data

**Extends:** L1Map(kdom, kcod, type = "L1\_C\_Product")

Property	Symbol	Type	Description
ms labels	$\ell_k$	array[L1Map] array[string]?	Maps to be composed. A list of labels for the maps

*Discussed in Section 20.11 (Product of maps)*

For a family of maps  $m_i : X \rightarrow Y_i$ , we define

$$\text{L1\_C\_Product}(\{m_i\}) : X \rightarrow \text{P\_C\_Product}(\{Y_i\})$$

### 26.4.16. L1\_C\_Series - Series composition

Data

**Extends:** L1Map(kdom, kcod, type = "L1\_C\_Series")

Property	Symbol	Type	Description
ms labels	$\ell_k$	array[L1Map] array[string]?	Maps to be composed. A list of labels for the maps

*Discussed in Section 20.12 (Series composition)*

### 26.4.17. L1\_C\_Intersection - Intersection

Data

**Extends:** `L1Map(kdom, kcod, type = "L1_C_Intersection")`

Property	Symbol	Type	Description
<code>ms</code> labels	$\ell_k$	<code>array[L1Map]</code> <code>array[string]?</code>	Maps to be composed. A list of labels for the maps

*Discussed in Section 20.13 (Union and Intersection of maps)*

For a family of maps  $m_i : X \rightarrow Y$ , we define

$$\begin{aligned} \text{L1\_C\_Intersection}(\{m_i\}) : X &\rightarrow Y \\ x &\mapsto \bigcap_i m_i(x) \end{aligned}$$

### 26.4.18. L1\_C\_Union - Union

Data

**Extends:** `L1Map(kdom, kcod, type = "L1_C_Union")`

Property	Symbol	Type	Description
<code>ms</code> labels	$\ell_k$	<code>array[L1Map]</code> <code>array[string]?</code>	Maps to be composed. A list of labels for the maps

*Discussed in Section 20.13 (Union and Intersection of maps)*

For a family of maps  $m_i : X \rightarrow Y$ , we define

$$\begin{aligned} \text{L1\_C\_Union}(\{m_i\}) : X &\rightarrow Y \\ x &\mapsto \bigcup_i m_i(x) \end{aligned}$$

### 26.4.19. L1\_C\_RefineDomain - Refines the domain of a monotone map.

Data

**Extends:** `L1Map(kdom, kcod, type = "L1_C_RefineDomain")`

Property	Symbol	Type	Description
<code>m</code>	$m$	<code>L1Map</code>	The map to be transformed.

### 26.4.20. L1\_C\_Trace - Trace

Data

**Extends:** `L1Map(kdom, kcod, type = "L1_C_Trace")`

Property	Symbol	Type	Description
<code>m</code>	$m$	<code>L1Map</code>	The map to be transformed.

*Discussed in Section 20.14 (Trace)*

This is the trace of the map.

#### 26.4.21. L1\_C\_WrapUnits - Decorates a map with units.

Data

**Extends:** `L1Map(kdom, kcod, type = "L1_C_WrapUnits")`

Property	Symbol	Type	Description
<code>m</code>	$m$	<code>L1Map</code>	The map to be transformed.
<code>kdom_units</code>		<code>Unit</code>	The units for the domain
<code>kcod_units</code>		<code>Unit</code>	The units for the codomain

#### 26.4.22. L1\_InvMul\_Opt - Finite-resolution optimistic approximation of the inverse of a multiplication map.

Data

**Extends:** `L1Map(kdom, kcod, type = "L1_InvMul_Opt")`

Property	Symbol	Type	Description
<code>n</code>		<code>integer</code>	Resolution (number of points in the produced antichain)
<code>ospace</code>		<code>Poset</code>	The poset in which the operation is performed.

This map provides an approximation of the map

$$x \mapsto \{\langle a, b \rangle \mid a \cdot b \leq x\} \quad (39)$$

The approximation is optimistic.

#### 26.4.23. L1\_InvMul\_Pes - Finite-resolution pessimistic approximation of the inverse of an addition map.

Data

**Extends:** `L1Map(kdom, kcod, type = "L1_InvMul_Pes")`

Property	Symbol	Type	Description
<code>n</code>		<code>integer</code>	Resolution (number of points in the produced antichain)
<code>ospace</code>		<code>Poset</code>	The poset in which the operation is performed.

This map provides an approximation of the map

$$x \mapsto \{\langle a, b \rangle \mid a \cdot b \leq x\} \quad (40)$$

The approximation is pessimistic.

#### 26.4.24. L1\_InvSum\_Opt - Finite-resolution optimistic approximation of the inverse of a multiplication map.

Data

**Extends:** `L1Map(kdom, kcod, type = "L1_InvSum_Opt")`

Property	Symbol	Type	Description
<code>n</code>		<code>integer</code>	Resolution (number of points in the produced antichain)
<code>ospace</code>		<code>Poset</code>	The poset in which the operation is performed.

This map provides an approximation of the map

$$x \mapsto \{\langle a, b \rangle \mid a + b \leq x\} \quad (41)$$

The approximation is optimistic.

#### 26.4.25. **L1\_InvSum\_Pes** - Finite-resolution pessimistic approximation of the inverse of an addition map.

Data

**Extends:** `L1Map(kdom, kcod, type = "L1_InvSum_Pes")`

Property	Symbol	Type	Description
<code>n</code>		integer	Resolution (number of points in the produced antichain)
<code>ospace</code>		Poset	The poset in which the operation is performed.

This map provides an approximation of the map

$$x \mapsto \{\langle a, b \rangle \mid a + b \leq x\} \quad (42)$$

The approximation is pessimistic.

#### 26.4.26. **L1\_FromFilter** - Filters based on a monotone map.

Data

**Extends:** `L1Map(kdom, kcod, type = "L1_FromFilter")`

Property	Symbol	Type	Description
<code>m</code>	<code>m</code>	<code>MonotoneMap</code>	A monotone map $m : \text{kdom} \rightarrow_{\text{Pos}} \text{Bool}$

*Discussed in Section 20.7 (Filtering)*

Defines the map:

$$x \mapsto \begin{cases} \{x\} & \text{if } m(x) \\ \emptyset & \text{otherwise} \end{cases} \quad (43)$$

#### 26.4.27. **L1\_L\_Linv** - Lower inverse of a monotone map

Data

**Extends:** `L1Map(kdom, kcod, type = "L1_L_Linv")`

Property	Symbol	Type	Description
<code>m</code>	<code>m</code>	<code>MonotoneMap</code>	The original map.

Creates the lower inverse of a monotone map.

#### 26.4.28. **L1\_Lift** - Lifts a monotone map

Data

**Extends:** `L1Map(kdom, kcod, type = "L1_Lift")`

Property	Symbol	Type	Description
<code>m</code>	<code>m</code>	<code>MonotoneMap</code>	A monotone map

*Discussed in Section 20.2 (Lifting maps)*

Lifts a monotone map to a `L1Map` in the obvious way:

$$\text{L1\_Lift}(m) : x \mapsto \{m(x)\} \quad (44)$$

#### 26.4.29. **L1\_TopAlternating** - Lower inverse for the meet map

Data

**Extends:** **L1Map**(**kdom**, **kcod**, type = "L1\_TopAlternating")

Property	Symbol	Type	Description
upper_bounds	$u_i^j$	array[array[any]]	

## 26.5. U1Map - Map to upper sets of resources.

Data

**Extends:** Root(version, description, hash, kind = "U1Map")

Property	Symbol	Type	Description
kdom	kdom	Poset	Kleisli domain of the map
kcod	kcod	Poset	Kleisli co-domain of the map
type		string	Discriminator variable to distinguish subtypes.

Discussed in Section 11.2 (**PosL** and **PosU**)

Subtypes based on the value for type

"U1_Constant"	Constant map
"U1_Entire"	Returns the entire poset
"U1_Explicit"	Map defined pointwise
"U1_Identity"	Lift of the identity map
"U1_Unknown"	Placeholder for an unknown map.
"U1_Catalog"	Map induced by a catalog of options.
"U1_IntersectionOfPrinUpperSets"	Intersection of principal upper sets.
"U1_RepresentPrincipalUpperSet"	Represent a principal upper set
"U1_UnionOfPrinUpperSets"	Union of principal upper sets.
"U1_C_CodSum"	Co-domain sum combination
"U1_C_CodSumSmash"	Co-domain (smash) sum combination
"U1_C_DomUnion"	Domain union
"U1_C_Parallel"	Monoidal product
"U1_C_ProdIntersection"	From product to intersection
"U1_C_Product"	Product
"U1_C_Series"	Series composition
"U1_C_Intersection"	Intersection
"U1_C_Union"	Union
"U1_C_RefineDomain"	Refines the domain of a monotone map.
"U1_C_Trace"	Trace
"U1_C_WrapUnits"	Decorates a map with units.
"U1_InvMul_Opt"	Finite-resolution optimistic approximation of the inverse of a multiplication map.
"U1_InvMul_Pes"	Finite-resolution pessimistic approximation of the inverse of a multiplication map.
"U1_InvSum_Opt"	Finite-resolution optimistic approximation of the inverse of an addition map.
"U1_InvSum_Pes"	Finite-resolution pessimistic approximation of the inverse of an addition map.
"U1_FromFilter"	Filters based on a monotone map.
"U1_L_Uinv"	Computes the upper inverse of a monotone map.
"U1_Lift"	Lifts a monotone map
"U1_Uinv_Join"	
"U1_Uinv_JoinConstant"	

### 26.5.1. U1\_Constant - Constant map

Data

**Extends:** U1Map(kdom, kcod, type = "U1\_Constant")

Property	Symbol	Type	Description
value	$U$	UpperSet	The constant value of the map, which is an upper set $U \subseteq \text{kcod}$ .

This is a constant map, which maps every element of the domain to the same value  $U$ .

### 26.5.2. **U1\_Entire** - Returns the entire poset

Data

**Extends:** `U1Map(kdom, kcod, type = "U1_Entire")`

This is the map defined as

$$x \mapsto \text{kcod}$$

which maps every element of the domain to the entire codomain poset `kcod`.

This is useful when the codomain doesn't have a compact representation in terms of antichains.

### 26.5.3. **U1\_Explicit** - Map defined pointwise

Data

**Extends:** `U1Map(kdom, kcod, type = "U1_Explicit")`

Property	Symbol	Type	Description
options		array[U1_Explicit_Option]	Option definitions

This is a map defined pointwise, where each option specifies a point in the domain and its corresponding value in the codomain.

#### **U1\_Explicit\_Option**

Data

Property	Symbol	Type	Description
x	$x$	any	A point in the domain of the map.
y	$y$	UpperSet	The upper set corresponding to the point $x$ in the domain.

### 26.5.4. **U1\_Identity** - Lift of the identity map

Data

**Extends:** `U1Map(kdom, kcod, type = "U1_Identity")`

*Discussed in Section 20.1 (Identity morphisms)*

The identity map:

$$x \mapsto \uparrow x$$

### 26.5.5. **U1\_Unknown** - Placeholder for an unknown map.

Data

**Extends:** `U1Map(kdom, kcod, type = "U1_Unknown")`

This is a placeholder for a map whose type is not known.



### 26.5.6. **U1\_Catalog** - Map induced by a catalog of options.

Data

**Extends:** `U1Map(kdom, kcod, type = "U1_Catalog")`

Property	Symbol	Type	Description
options	$\langle f_k, r_k^* \rangle$	array[U1_Catalog_Options]	The options in the catalog

*Discussed in Section 20.3 (Catalog maps)*

This map is defined by a catalog of options  $\llbracket \langle f_k, r_k^* \rangle \rrbracket$ :

$$f^* \mapsto \bigcup_k \{ \uparrow r_k \mid f_k \leq x \} \quad (45)$$

**U1\_Catalog\_Options** - An option in the catalog

Data

Property	Symbol	Type	Description
f	$f$	any	A point in the domain.
r	$r^*$	any	A point in the codomain

An option in the catalog, where  $f$  is a point in the domain and  $r$  is the corresponding upper set in the codomain.

### 26.5.7. **U1\_IntersectionOfPrinUpperSets** - Intersection of principal upper sets.

Data

**Extends:** `U1Map(kdom, kcod, type = "U1_IntersectionOfPrinUpperSets")`

*Discussed in Section 20.4 (Union and intersection of principal lower sets)*

The domain `kdom` must be a product poset.

This map is defined as follows:

$$\langle x_i \rangle_i \mapsto \bigcap_i \uparrow x_i \quad (46)$$

### 26.5.8. U1\_RepresentPrincipalUpperSet - Represent a principal upper set

Data

**Extends:** `U1Map(kdom, kcod, type = "U1_RepresentPrincipalUpperSet")`

*Discussed in Section 20.5 (Representing principal lower and upper sets)*

This map is defined when `kdom` and `kcod` are subsets of a common ambient poset  $P$ .

It is defined as:

$$x \mapsto \max\{S \in \text{P\_C\_UpperSets}(\text{kcod}) \mid S \subseteq \uparrow_p x\} \quad (47)$$

The idea is that we want to “represent” the principal upper set  $\uparrow_p x$  in terms of the codomain `kcod`.

For example, consider the case

- `kdom` =  $2\mathbb{Z}$ , the even integers
- `kcod` =  $3\mathbb{Z}$ , the multiples of 3

They are both subposets of the ambient poset  $\mathbb{Z}$ . The map would associate each even integer  $x$  with the up closure of the smallest multiple of 3 greater than or equal to  $x$ :

$$\begin{aligned} 10 &\mapsto \uparrow 12 \\ 8 &\mapsto \uparrow 9 \\ 6 &\mapsto \uparrow 6 \\ 4 &\mapsto \uparrow 6 \\ 2 &\mapsto \uparrow 3 \\ 0 &\mapsto \uparrow 0 \end{aligned}$$

In general, the posets are not total orders, so the result is not necessarily a principal upper set.

### 26.5.9. U1\_UnionOfPrinUpperSets - Union of principal upper sets.

Data

**Extends:** `U1Map(kdom, kcod, type = "U1_UnionOfPrinUpperSets")`

*Discussed in Section 20.4 (Union and intersection of principal lower sets)*

The domain `kdom` must be a product poset.

This map is defined as follows:

$$\langle x_i \rangle_i \mapsto \bigcup_i \uparrow x_i \quad (48)$$

### 26.5.10. U1\_C\_CodSum - Co-domain sum combination

Data

**Extends:** `U1Map(kdom, kcod, type = "U1_C_CodSum")`

Property	Symbol	Type	Description
<code>ms</code> labels	$m_i$	array[ <code>U1Map</code> ] array[string]?	Maps to be composed. A list of labels for the maps

*Discussed in Section 20.10 (Codomain Sum)*

Given two maps  $f : A \rightarrow B$  and  $g : A \rightarrow C$ , the codomain sum of these maps is a map that combines the codomains of both maps:

$$\text{U1\_C\_CodSum}(f, g) : A \rightarrow \text{P\_C\_Sum}(B, C) \quad (49)$$

It is defined as follows:

$$x \mapsto \{\text{in}_1(b) \mid b \in f(x)\} \cup \{\text{in}_2(c) \mid c \in g(x)\} \quad (50)$$

### 26.5.11. **U1\_C\_CodSumSmash** - Co-domain (smash) sum combination

Data

**Extends:** `U1Map(kdom, kcod, type = "U1_C_CodSumSmash")`

Property	Symbol	Type	Description
<code>ms</code> labels	$m_i$	array[U1Map] array[string]?	Maps to be composed. A list of labels for the maps

Given two maps  $f : A \rightarrow B$  and  $g : A \rightarrow C$ , the codomain sum of these maps is a map that combines the codomains of both maps:

$$\text{U1\_C\_CodSumSmash}(f, g) : A \rightarrow \text{P\_C\_SumSmash}(B, C) \quad (51)$$

It is defined as follows:

$$x \mapsto \{\text{in}_1(b) \mid b \in f(x)\} \cup \{\text{in}_2(c) \mid c \in g(x)\} \quad (52)$$

### 26.5.12. **U1\_C\_DomUnion** - Domain union

Data

**Extends:** `U1Map(kdom, kcod, type = "U1_C_DomUnion")`

Property	Symbol	Type	Description
<code>ms</code> labels	$m_i$	array[U1Map] array[string]?	Maps to be composed. A list of labels for the maps

This is the equivalent of `M_C_DomUnion`.

Given two maps  $f : A \rightarrow B$  and  $g : C \rightarrow B$ , the domain union of these maps is a map that combines the domains of both maps:

$$\text{U1\_C\_DomUnion}(f, g) : \text{P\_F\_C\_Union}(A, C) \rightarrow B \quad (53)$$

The value is defined as follows:

$$\text{U1\_C\_DomUnion}(f, g) : x \mapsto \begin{cases} f(x) & \text{if } x \in A \\ g(x) & \text{if } x \in C \end{cases} \quad (54)$$

Note that the order of the maps does matter. We use the first map whose domain contains the input.

### 26.5.13. **U1\_C\_Parallel** - Monoidal product

Data

**Extends:** `U1Map(kdom, kcod, type = "U1_C_Parallel")`

Property	Symbol	Type	Description
<code>ms</code> labels	$m_i$	array[U1Map] array[string]?	Maps to be composed. A list of labels for the maps

*Discussed in Section 20.8 (Parallel composition)*

For a family of maps  $m_i : X_i \rightarrow Y_i$ , the monoidal product is defined as:

$$\text{U1\_C\_Parallel}(\{m_i\}) : \text{P\_C\_Product}(\{X_i\}) \rightarrow \text{P\_C\_Product}(\{Y_i\})$$

#### 26.5.14. **U1\_C\_ProdIntersection** - From product to intersection

Data

**Extends:** **U1Map**(**kdom**, **kcod**, type = "U1\_C\_ProdIntersection")

Property	Symbol	Type	Description
<b>ms</b> labels	$m_i$	array[ <b>U1Map</b> ] array[string]?	Maps to be composed. A list of labels for the maps

The domain **kdom** should be a product. The map is defined as follows:

$$\langle x_i \rangle_i \mapsto \bigcap_i f_i(x_i) \quad (55)$$

#### 26.5.15. **U1\_C\_Product** - Product

Data

**Extends:** **U1Map**(**kdom**, **kcod**, type = "U1\_C\_Product")

Property	Symbol	Type	Description
<b>ms</b> labels	$m_i$	array[ <b>U1Map</b> ] array[string]?	Maps to be composed. A list of labels for the maps

*Discussed in Section 20.11 (Product of maps)*

For a family of maps  $m_i : X \rightarrow Y_i$ , we define

$$\mathbf{U1\_C\_Product}(\{m_i\}) : X \rightarrow \mathbf{P\_C\_Product}(\{Y_i\})$$

#### 26.5.16. **U1\_C\_Series** - Series composition

Data

**Extends:** **U1Map**(**kdom**, **kcod**, type = "U1\_C\_Series")

Property	Symbol	Type	Description
<b>ms</b> labels	$m_i$	array[ <b>U1Map</b> ] array[string]?	Maps to be composed. A list of labels for the maps

*Discussed in Section 20.12 (Series composition)*

#### 26.5.17. **U1\_C\_Intersection** - Intersection

Data

**Extends:** **U1Map**(**kdom**, **kcod**, type = "U1\_C\_Intersection")

Property	Symbol	Type	Description
<b>ms</b> labels	$m_i$	array[ <b>U1Map</b> ] array[string]?	Maps to be composed. A list of labels for the maps

*Discussed in Section 20.13 (Union and Intersection of maps)*

For a family of maps  $m_i : X \rightarrow Y$ , we define

$$\mathbf{U1\_C\_Intersection}(\{m_i\}) : X \rightarrow Y$$

$$x \mapsto \bigcap_i m_i(x)$$

### 26.5.18. **U1\_C\_Union** - Union

Data

**Extends:** **U1Map**(**kdom**, **kcod**, type = "U1\_C\_Union")

Property	Symbol	Type	Description
<b>m</b> s labels	$m_i$	array[ <b>U1Map</b> ] array[string]?	Maps to be composed. A list of labels for the maps

*Discussed in Section 20.13 (Union and Intersection of maps)*

For a family of maps  $m_i : X \rightarrow Y$ , we define

$$\begin{aligned} \mathbf{U1\_C\_Union}(\{m_i\}) : X \rightarrow Y \\ x \mapsto \bigcup_i m_i(x) \end{aligned}$$

### 26.5.19. **U1\_C\_RefineDomain** - Refines the domain of a monotone map.

Data

**Extends:** **U1Map**(**kdom**, **kcod**, type = "U1\_C\_RefineDomain")

Property	Symbol	Type	Description
<b>m</b>	$m$	<b>U1Map</b>	Original map

### 26.5.20. **U1\_C\_Trace** - Trace

Data

**Extends:** **U1Map**(**kdom**, **kcod**, type = "U1\_C\_Trace")

Property	Symbol	Type	Description
<b>m</b>	$m$	<b>U1Map</b>	Original map

*Discussed in Section 20.14 (Trace)*

This is the trace of the map.

### 26.5.21. **U1\_C\_WrapUnits** - Decorates a map with units.

Data

**Extends:** **U1Map**(**kdom**, **kcod**, type = "U1\_C\_WrapUnits")

Property	Symbol	Type	Description
<b>m</b> kdom_units kcod_units	$m$	<b>U1Map</b> Unit Unit	Original map Units for the domain Units for the codomain

### 26.5.22. **U1\_InvMul\_Opt** - Finite-resolution optimistic approximation of the inverse of a multiplication map.

Data

**Extends:** `U1Map(kdom, kcod, type = "U1_InvMul_Opt")`

Property	Symbol	Type	Description
n ospace		integer Poset	Resolution (number of points in the produced antichain) The poset in which the operation is performed.

This map provides an approximation of the map

$$x \mapsto \{\langle a, b \rangle \mid a \cdot b \geq x\} \quad (56)$$

The approximation is optimistic.

### 26.5.23. **U1\_InvMul\_Pes** - Finite-resolution pessimistic approximation of the inverse of a multiplication map.

Data

**Extends:** `U1Map(kdom, kcod, type = "U1_InvMul_Pes")`

Property	Symbol	Type	Description
n ospace		integer Poset	Resolution (number of points in the produced antichain) The poset in which the operation is performed.

This map provides an approximation of the map

$$x \mapsto \{\langle a, b \rangle \mid a \cdot b \geq x\} \quad (57)$$

The approximation is pessimistic.

### 26.5.24. **U1\_InvSum\_Opt** - Finite-resolution optimistic approximation of the inverse of an addition map.

Data

**Extends:** `U1Map(kdom, kcod, type = "U1_InvSum_Opt")`

Property	Symbol	Type	Description
n ospace		integer Poset	Resolution (number of points in the produced antichain) The poset in which the operation is performed.

*Discussed in Section 20.6 (Generic inverses for mathematical operations)*

This map provides an approximation of the map

$$x \mapsto \{\langle a, b \rangle \mid a + b \geq x\} \quad (58)$$

The approximation is optimistic.

### 26.5.25. **U1\_InvSum\_Pes** - Finite-resolution pessimistic approximation of the inverse of an addition map.

Data

**Extends:** `U1Map(kdom, kcod, type = "U1_InvSum_Pes")`

Property	Symbol	Type	Description
n ospace		integer Poset	Resolution (number of points in the produced antichain) The poset in which the operation is performed.

*Discussed in Section 20.6 (Generic inverses for mathematical operations)*

This map provides an approximation of the map

$$x \mapsto \{\langle a, b \rangle \mid a + b \geq x\} \quad (59)$$

The approximation is pessimistic.

### 26.5.26. **U1\_FromFilter** - Filters based on a monotone map.

Data

**Extends:** `U1Map(kdom, kcod, type = "U1_FromFilter")`

Property	Symbol	Type	Description
<code>m</code>	<code>m</code>	<code>MonotoneMap</code>	A monotone map $m : \text{kdom} \rightarrow \text{Bool}$

*Discussed in Section 20.7 (Filtering)*

Defines the map:

$$x \mapsto \begin{cases} \{x\} & \text{if } m(x) \\ \emptyset & \text{otherwise} \end{cases} \quad (60)$$

### 26.5.27. **U1\_L\_Uinv** - Computes the upper inverse of a monotone map.

Data

**Extends:** `U1Map(kdom, kcod, type = "U1_L_Uinv")`

Property	Symbol	Type	Description
<code>m</code>	<code>m</code>	<code>MonotoneMap</code>	A monotone map $m : \text{kdom} \rightarrow \text{kcod}$

Computes the upper inverse  $\text{Uinv}(m)$  of the monotone map  $m$ .

For a monotone map  $m : P \rightarrow Q$ , the upper inverse is defined as:

$$(\text{Uinv } f) : Q^{\text{op}} \rightarrow_{\text{pos}} P\_C\_UpperSets(P) \quad (61)$$

$$q \mapsto \{p \in P \mid q \leq f(p)\} \quad (62)$$

### 26.5.28. **U1\_Lift** - Lifts a monotone map

Data

**Extends:** `U1Map(kdom, kcod, type = "U1_Lift")`

Property	Symbol	Type	Description
<code>m</code>	<code>m</code>	<code>MonotoneMap</code>	A monotone map

*Discussed in Section 20.2 (Lifting maps)*

Lifts a monotone map to a **U1Map** in the obvious way:

$$\text{U1\_Lift}(m) : x \mapsto \uparrow m(x) \quad (63)$$

### 26.5.29. **U1\_Uinv\_Join**

Data

**Extends:** `U1Map(kdom, kcod, type = "U1_Uinv_Join")`

Property	Symbol	Type	Description
<code>lower_bounds</code>		<code>array[array[any]]</code>	

26.5.30. **U1\_Uinv\_JoinConstant**

Data

Extends: <b>U1Map</b> ( <b>kdom</b> , <b>kcod</b> , type = "U1_Uinv_JoinConstant")			
Property	Symbol	Type	Description
join1_dom value		Poset Value	



## 26.6. LMap - Map to lower sets of functionalities and implementations.

Data

**Extends:** Root(version, description, hash, kind = "LMap")

Property	Symbol	Type	Description
kdom	kdom	Poset	The Kleisli domain of the map.
kcod	kcod	Poset	The Kleisli codomain of the map.
kimp	kimp	Poset	The implementation poset of the map.
type		string	Discriminator variable to distinguish subtypes.

*Discussed in Section 14.3 (Categories **PosUI** and **PosLI**)*

Subtypes based on the value for type

"L_Constant"	Constant map
"L_Identity"	Identity morphism
"L_Unknown"	Placeholder for an unknown map
"L_Catalog"	LMap for a catalog
"L_C_Parallel"	Monoidal product
"L_C_Series"	Series composition
"L_C_Intersection"	Intersection of maps
"L_C_Union"	Union of maps
"L_C_ITransform"	Transforms the implementation of another map.
"L_C_RefineDomain"	Refines the domain of a monotone map
"L_C_Trace"	Trace
"L_C_WrapUnits"	Decorates a map with units.
"L_L_Lift1_Constant"	Lifts a L1Map morphisms with a constant value for the implementation.
"L_L_Lift1_Transform"	Lifts a L1Map morphism with a function to compute the implementation.

### 26.6.1. L\_Constant - Constant map

Data

**Extends:** LMap(kdom, kcod, kimp, type = "L\_Constant")

Property	Symbol	Type	Description
value		LowerSet	The lower set that is the value of the constant map.

*Discussed in Section 21.2 (Constant maps)*

### 26.6.2. L\_Identity - Identity morphism

Data

**Extends:** LMap(kdom, kcod, kimp, type = "L\_Identity")

*Discussed in Section 21.1 (Identity)*

### 26.6.3. L\_Unknown - Placeholder for an unknown map

Data

**Extends:** LMap(kdom, kcod, kimp, type = "L\_Unknown")

#### 26.6.4. **L\_Catalog** - LMap for a catalog

Data

**Extends:** `LMap(kdom, kcod, kimp, type = "L_Catalog")`

Property	Symbol	Type	Description
options		array[L_Catalog_Options]	

*Discussed in Section 21.3 (Catalog maps)*

This is the **LMap** that arises from a **DP\_Catalog**.

#### **L\_Catalog\_Options** - Options for **L\_Catalog**

Data

Property	Symbol	Type	Description
f	$f$	any	
r	$r^*$	any	
i	$i$	any	

#### 26.6.5. **L\_C\_Parallel** - Monoidal product

Data

**Extends:** `LMap(kdom, kcod, kimp, type = "L_C_Parallel")`

Property	Symbol	Type	Description
ms		array[LMap]	Maps to be composed.
labels		array[string]?	Labels for the maps.

*Discussed in Section 21.6 (Parallel composition)*

#### 26.6.6. **L\_C\_Series** - Series composition

Data

**Extends:** `LMap(kdom, kcod, kimp, type = "L_C_Series")`

Property	Symbol	Type	Description
ms		array[LMap]	Maps to be composed.
labels		array[string]?	Labels for the maps.

*Discussed in Section 21.5 (Series composition)*

#### 26.6.7. **L\_C\_Intersection** - Intersection of maps

Data

**Extends:** `LMap(kdom, kcod, kimp, type = "L_C_Intersection")`

Property	Symbol	Type	Description
ms		array[LMap]	Maps to be composed.
labels		array[string]?	Labels for the maps.

*Discussed in Section 21.7 (Intersection of maps)*

### 26.6.8. **L\_C\_Union** - Union of maps

Data

**Extends:** `LMap(kdom, kcod, kimp, type = "L_C_Union")`

Property	Symbol	Type	Description
<code>ms</code>		<code>array[LMap]</code>	Maps to be composed.
<code>labels</code>		<code>array[string]?</code>	Labels for the maps.

*Discussed in Section 21.8 (Union of maps)*

### 26.6.9. **L\_C\_ITransform** - Transforms the implementation of another map.

Data

**Extends:** `LMap(kdom, kcod, kimp, type = "L_C_ITransform")`

Property	Symbol	Type	Description
<code>m</code>	$m$	<code>LMap</code>	The map to be transformed
<code>transform</code>	$T$	<code>MonotoneMap</code>	The transformation to apply to the implementation of the map.

*Discussed in Section 21.9 (Transforming maps)*

### 26.6.10. **L\_C\_RefineDomain** - Refines the domain of a monotone map

Data

**Extends:** `LMap(kdom, kcod, kimp, type = "L_C_RefineDomain")`

Property	Symbol	Type	Description
<code>m</code>	$m$	<code>LMap</code>	The map to be transformed

### 26.6.11. **L\_C\_Trace** - Trace

Data

**Extends:** `LMap(kdom, kcod, kimp, type = "L_C_Trace")`

Property	Symbol	Type	Description
<code>m</code>	$m$	<code>LMap</code>	The map to be transformed
<code>m_proj</code>	$m'$	<code>L1Map</code>	The projection of the LMap $m$ to a L1Map.

*Discussed in Section 21.10 (Trace)*

This is the trace of a `LMap`  $m$ .

Because of computation convenience, we also require the projection  $m'$  of the `LMap`  $m$  to a `L1Map`.

### 26.6.12. **L\_C\_WrapUnits** - Decorates a map with units.

Data

**Extends:** `LMap(kdom, kcod, kimp, type = "L_C_WrapUnits")`

Property	Symbol	Type	Description
<code>m</code>	$m$	<code>LMap</code>	The map to be transformed
<code>kdom_units</code>		<code>Unit</code>	The units for decorating <code>kdom</code>
<code>kcod_units</code>		<code>Unit</code>	The units for decorating <code>kcod</code>
<code>kimp_units</code>		<code>Unit</code>	The units for decorating <code>kimp</code>

### 26.6.13. `L_Lift1_Constant` - Lifts a `L1Map` morphisms with a constant value for the implementation.

**Extends:** `LMap(kdom, kcod, kimp, type = "L_Lift1_Constant")`

Property	Symbol	Type	Description
<code>m</code> value	$m$ $v$	<code>L1Map</code> any	A <code>L1Map</code> morphism. A constant value for the implementation of the <code>L1Map</code> morphism.

*Discussed in Section 21.4 (Lifting maps)*

### 26.6.14. `L_Lift1_Transform` - Lifts a `L1Map` morphism with a function to compute the implementation.

**Extends:** `LMap(kdom, kcod, kimp, type = "L_Lift1_Transform")`

Property	Symbol	Type	Description
<code>m</code> <code>transform</code>	$\ell$ $t$	<code>L1Map</code> <code>MonotoneMap</code>	A <code>L1Map</code> morphism. A function to compute the implementation of the <code>L1Map</code> morphism.

*Discussed in Section 21.4 (Lifting maps)*

## 26.7. UMap - Map to upper sets of resources and implementations.

Data

**Extends:** Root(version, description, hash, kind = "UMap")

Property	Symbol	Type	Description
kdom	kdom	Poset	The Kleisli domain of the map.
kcod	kcod	Poset	The Kleisli codomain of the map.
kimp	kimp	Poset	The implementation poset of the map.
type		string	Discriminator variable to distinguish subtypes.

Discussed in Section 14.3 (Categories **PosUI** and **PosLI**)

Subtypes based on the value for type

"U_Constant"	Constant map
"U_Identity"	Identity
"U_Unknown"	Placeholder for an unknown map
"U_Catalog"	UMap for a catalog
"U_C_Parallel"	Monoidal product
"U_C_Series"	Series composition
"U_C_Intersection"	Intersection of maps
"U_C_Union"	Union of maps
"U_C_ITransform"	Transforms the implementation of another map.
"U_C_RefineDomain"	Refines the domain of a monotone map
"U_C_Trace"	Trace
"U_C_WrapUnits"	Decorates a map with units.
"U_L_Lift1_Constant"	Lifts a U1Map morphism with a constant value for the implementation.
"U_L_Lift1_Transform"	Lifts a U1Map morphism with a function to compute the implementation.

### 26.7.1. U\_Constant - Constant map

Data

**Extends:** UMap(kdom, kcod, kimp, type = "U\_Constant")

Property	Symbol	Type	Description
value		UpperSet	The upper set that is the value of the constant map.

Discussed in Section 21.2 (Constant maps)

### 26.7.2. U\_Identity - Identity

Data

**Extends:** UMap(kdom, kcod, kimp, type = "U\_Identity")

Discussed in Section 21.1 (Identity)

### 26.7.3. U\_Unknown - Placeholder for an unknown map

Data

**Extends:** UMap(kdom, kcod, kimp, type = "U\_Unknown")

#### 26.7.4. **U\_Catalog** - UMap for a catalog

Data

**Extends:** `UMap(kdom, kcod, kimp, type = "U_Catalog")`

Property	Symbol	Type	Description
options		array[U_Catalog_Options]	The options in the catalog.

*Discussed in Section 21.3 (Catalog maps)*

This is the **UMap** that arises from a **DP\_Catalog**.

#### **U\_Catalog\_Options** - An option in the catalog

Data

Property	Symbol	Type	Description
f	$f$	any	
r	$r^*$	any	
i	$i$	any	

An option in the catalog.

#### 26.7.5. **U\_C\_Parallel** - Monoidal product

Data

**Extends:** `UMap(kdom, kcod, kimp, type = "U_C_Parallel")`

Property	Symbol	Type	Description
ms		array[UMap]	Maps to be composed.
labels		array[string]?	Labels for the maps.

*Discussed in Section 21.6 (Parallel composition)*

#### 26.7.6. **U\_C\_Series** - Series composition

Data

**Extends:** `UMap(kdom, kcod, kimp, type = "U_C_Series")`

Property	Symbol	Type	Description
ms		array[UMap]	Maps to be composed.
labels		array[string]?	Labels for the maps.

*Discussed in Section 21.5 (Series composition)*

#### 26.7.7. **U\_C\_Intersection** - Intersection of maps

Data

**Extends:** `UMap(kdom, kcod, kimp, type = "U_C_Intersection")`

Property	Symbol	Type	Description
ms		array[UMap]	Maps to be composed.
labels		array[string]?	Labels for the maps.

*Discussed in Section 21.7 (Intersection of maps)*

### 26.7.8. **U\_C\_Union** - Union of maps

Data

Extends: `UMap(kdom, kcod, kimp, type = "U_C_Union")`

Property	Symbol	Type	Description
<code>ms</code> <code>labels</code>		<code>array[UMap]</code> <code>array[string]?</code>	Maps to be composed. Labels for the maps.

*Discussed in Section 21.8 (Union of maps)*

### 26.7.9. **U\_C\_ITransform** - Transforms the implementation of another map.

Data

Extends: `UMap(kdom, kcod, kimp, type = "U_C_ITransform")`

Property	Symbol	Type	Description
<code>m</code> <code>transform</code>	<code>u</code>	<code>UMap</code> <code>MonotoneMap</code>	The original map

*Discussed in Section 21.9 (Transforming maps)*

### 26.7.10. **U\_C\_RefinedDomain** - Refines the domain of a monotone map

Data

Extends: `UMap(kdom, kcod, kimp, type = "U_C_RefinedDomain")`

Property	Symbol	Type	Description
<code>m</code>	<code>u</code>	<code>UMap</code>	The original map

### 26.7.11. **U\_C\_Trace** - Trace

Data

Extends: `UMap(kdom, kcod, kimp, type = "U_C_Trace")`

Property	Symbol	Type	Description
<code>m</code> <code>m_proj</code>	<code>u</code> <code>m'</code>	<code>UMap</code> <code>U1Map</code>	The original map The projection of the <code>UMap m</code> .

*Discussed in Section 21.10 (Trace)*

This is the trace of a `UMap m`.

Because of computation convenience, we also require the projection `m'` of the `UMap m` to a `U1Map`.

### 26.7.12. **U\_C\_WrapUnits** - Decorates a map with units.

Data

Extends: `UMap(kdom, kcod, kimp, type = "U_C_WrapUnits")`

Property	Symbol	Type	Description
<code>m</code> <code>kdom_units</code> <code>kcod_units</code> <code>kimp_units</code>	<code>u</code>	<code>UMap</code> <code>Unit</code> <code>Unit</code> <code>Unit</code>	The original map The units for decorating <code>kdom</code> The units for decorating <code>kcod</code> The units for decorating <code>kimp</code>

**26.7.13. `U_L_Lift1_Constant` - Lifts a `U1Map` morphism with a constant value for the implementation.**

Extends: <code>UMap(kdom, kcod, kimp, type = "U_L_Lift1_Constant")</code>			
Property	Symbol	Type	Description
<code>m</code> value	$u$ $v$	<code>U1Map</code> any	A <code>U1Map</code> morphism. A constant value for the implementation of the <code>U1Map</code> morphism.

*Discussed in Section 21.4 (Lifting maps)*

**26.7.14. `U_L_Lift1_Transform` - Lifts a `U1Map` morphism with a function to compute the implementation.**

Extends: <code>UMap(kdom, kcod, kimp, type = "U_L_Lift1_Transform")</code>			
Property	Symbol	Type	Description
<code>m</code> <code>transform</code>	$m$ $f$	<code>U1Map</code> <code>MonotoneMap</code>	A <code>U1Map</code> morphism. A function that computes the implementation of the <code>U1Map</code> morphism.

*Discussed in Section 21.4 (Lifting maps)*



## 26.8. SL1Map - Scalable map to lower sets of functionalities.

Data	<b>Extends:</b> Root(version, description, hash, kind = "SL1Map")			
	Property	Symbol	Type	Description
	kdom	kdom	Poset	The Kleisli domain of the map.
	kcod	kcod	Poset	The Kleisli codomain of the map.
	pes	$S^{\ominus}$	Poset	The resolution poset for pessimistic estimate.
	opt	$S^{\oplus}$	Poset	The resolution poset for optimistic estimate.
	type		string	Discriminator variable to distinguish subtypes.

### Subtypes based on the value for type

"SL1_C_Parallel"	Monoidal product
"SL1_C_Series"	Series composition
"SL1_Identity"	Identity
"SL1_Unknown"	Placeholder for an unknown SL1Map
"SL1_C_CodSum"	Sum of maps
"SL1_C_CodSumSmash"	Smash sum
"SL1_C_ProdIntersection"	Product of domains, intersection of codomains
"SL1_C_Product"	Product of SL1 maps
"SL1_C_Intersection"	Intersection of SL1 maps
"SL1_C_Union"	Union of SL1 maps
"SL1_C_RefineDomain"	Refinement of the domain
"SL1_C_Trace"	Trace
"SL1_C_WrapUnits"	Decorates a map with units for the domain and codomain.
"SL1_Exact"	Lifts a L1Map to a SL1Map.
"SL1_InvMultiply"	The lower inverse of multiplication.
"SL1_InvSum"	The lower inverse of addition.
"SL1_C_ExplicitApprox"	Constructs a SL1Map from explicit approximations of L1Map maps.

### 26.8.1. SL1\_C\_Parallel - Monoidal product

Data	<b>Extends:</b> SL1Map(kdom, kcod, pes, opt, type = "SL1_C_Parallel")			
	Property	Symbol	Type	Description
	ms labels	$\ell_k$	array[SL1Map] array[string]?	Maps to be composed. A list of labels for the maps.

*Discussed in Section 22.3 (Parallel composition)*

This is the generalization of L1\_C\_Parallel1.

### 26.8.2. SL1\_C\_Series - Series composition

Data	<b>Extends:</b> SL1Map(kdom, kcod, pes, opt, type = "SL1_C_Series")			
	Property	Symbol	Type	Description
	ms labels	$\ell_k$	array[SL1Map] array[string]?	Maps to be composed. A list of labels for the maps.

*Discussed in Section 22.4 (Series composition)*

This is the generalization of L1\_C\_Series.

### 26.8.3. **SL1\_Identity** - Identity

Data

**Extends:** `SL1Map(kdom, kcod, pes, opt, type = "SL1_Identity")`

*Discussed in Section 22.1 (Identities)*

### 26.8.4. **SL1\_Unknown** - Placeholder for an unknown SL1Map

Data

**Extends:** `SL1Map(kdom, kcod, pes, opt, type = "SL1_Unknown")`

### 26.8.5. **SL1\_C\_CodSum** - Sum of maps

Data

**Extends:** `SL1Map(kdom, kcod, pes, opt, type = "SL1_C_CodSum")`

Property	Symbol	Type	Description
<code>ms</code> labels	$\ell_k$	array[ <code>SL1Map</code> ] array[string]?	Maps to be composed. A list of labels for the maps.

*Discussed in Section 22.9 (Sum)*

This is the generalization of `L1_C_CodSum`.

### 26.8.6. **SL1\_C\_CodSumSmash** - Smash sum

Data

**Extends:** `SL1Map(kdom, kcod, pes, opt, type = "SL1_C_CodSumSmash")`

Property	Symbol	Type	Description
<code>ms</code> labels	$\ell_k$	array[ <code>SL1Map</code> ] array[string]?	Maps to be composed. A list of labels for the maps.

*Discussed in Section 22.9 (Sum)*

This is the generalization of `L1_C_CodSumSmash`.

### 26.8.7. **SL1\_C\_ProdIntersection** - Product of domains, intersection of codomains

Data

**Extends:** `SL1Map(kdom, kcod, pes, opt, type = "SL1_C_ProdIntersection")`

Property	Symbol	Type	Description
<code>ms</code> labels	$\ell_k$	array[ <code>SL1Map</code> ] array[string]?	Maps to be composed. A list of labels for the maps.

*Discussed in Section 22.10 (Product intersection)*

This is the generalization of `L1_C_ProdIntersection`.

### 26.8.8. **SL1\_C\_Product** - Product of SL1 maps

Data

**Extends:** **SL1Map**(**kdom**, **kcod**, pes, opt, type = "SL1\_C\_Product")

Property	Symbol	Type	Description
<b>ms</b> labels	$\ell_k$	array[ <b>SL1Map</b> ] array[string]?	Maps to be composed. A list of labels for the maps.

*Discussed in Section 22.8 (Product)*

This is the generalization of **L1\_C\_Product**.

### 26.8.9. **SL1\_C\_Intersection** - Intersection of SL1 maps

Data

**Extends:** **SL1Map**(**kdom**, **kcod**, pes, opt, type = "SL1\_C\_Intersection")

Property	Symbol	Type	Description
<b>ms</b> labels	$\ell_k$	array[ <b>SL1Map</b> ] array[string]?	Maps to be composed. A list of labels for the maps.

*Discussed in Section 22.6 (Intersection)*

This is the generalization of **L1\_C\_Intersection**.

### 26.8.10. **SL1\_C\_Union** - Union of SL1 maps

Data

**Extends:** **SL1Map**(**kdom**, **kcod**, pes, opt, type = "SL1\_C\_Union")

Property	Symbol	Type	Description
<b>ms</b> labels	$\ell_k$	array[ <b>SL1Map</b> ] array[string]?	Maps to be composed. A list of labels for the maps.

*Discussed in Section 22.5 (Union)*

This is the generalization of **L1\_C\_Union**.

### 26.8.11. **SL1\_C\_RefineDomain** - Refinement of the domain

Data

**Extends:** **SL1Map**(**kdom**, **kcod**, pes, opt, type = "SL1\_C\_RefineDomain")

Property	Symbol	Type	Description
<b>m</b>	$m$	<b>SL1Map</b>	The map to be transformed

This is the generalization of **L1\_C\_RefineDomain**.

### 26.8.12. **SL1\_C\_Trace** - Trace

Data

**Extends:** **SL1Map**(**kdom**, **kcod**, pes, opt, type = "SL1\_C\_Trace")

Property	Symbol	Type	Description
<b>m</b>	<i>m</i>	<b>SL1Map</b>	The map to be transformed

*Discussed in Section 22.7 (Trace)*

This is the generalization of **L1\_C\_Trace**.

### 26.8.13. **SL1\_C\_WrapUnits** - Decorates a map with units for the domain and codomain.

Data

**Extends:** **SL1Map**(**kdom**, **kcod**, pes, opt, type = "SL1\_C\_WrapUnits")

Property	Symbol	Type	Description
<b>m</b>	<i>m</i>	<b>SL1Map</b>	The map to be transformed
<b>kdom_units</b>		<b>Unit</b>	The units for the domain.
<b>kcod_units</b>		<b>Unit</b>	The units for the codomain.

This is the generalization of **L1\_C\_WrapUnits**.

### 26.8.14. **SL1\_Exact** - Lifts a L1Map to a SL1Map.

Data

**Extends:** **SL1Map**(**kdom**, **kcod**, pes, opt, type = "SL1\_Exact")

Property	Symbol	Type	Description
<b>m</b>	$\ell$	<b>L1Map</b>	The L1Map to lift.

*Discussed in Section 22.2 (Lifting)*

### 26.8.15. **SL1\_InvMultiply** - The lower inverse of multiplication.

Data

**Extends:** **SL1Map**(**kdom**, **kcod**, pes, opt, type = "SL1\_InvMultiply")

Property	Symbol	Type	Description
<b>ospace</b>	<i>P</i>	<b>Poset</b>	The poset where the operation takes place

*Discussed in Section 22.11 (Scalable inverse of sum and multiplication operations)*

### 26.8.16. **SL1\_InvSum** - The lower inverse of addition.

Data

**Extends:** **SL1Map**(**kdom**, **kcod**, pes, opt, type = "SL1\_InvSum")

Property	Symbol	Type	Description
<b>ospace</b>	<i>P</i>	<b>Poset</b>	The poset where the operation takes place

*Discussed in Section 22.11 (Scalable inverse of sum and multiplication operations)*

### 26.8.17. **SL1\_C\_ExplicitApprox** - Constructs a SL1Map from explicit approximations of L1Map maps.

Data

**Extends:** **SL1Map**(**kdom**, **kcod**, **pes**, **opt**, **type** = "SL1\_C\_ExplicitApprox")

Property	Symbol	Type	Description
<b>optimistic</b>		array[ <b>L1Map</b> ]	The optimistic approximations of the L1Map.
<b>optimistic_labels</b>		array[string]?	Labels for the optimistic approximations.
<b>pessimistic</b>		array[ <b>L1Map</b> ]	The pessimistic approximations of the L1Map.
<b>pessimistic_labels</b>		array[string]?	Labels for the pessimistic approximations.

*Discussed in Section 22.12 (Explicit approximation)*

## 26.9. SU1Map - Scalable map to upper sets of resources.

Data	<b>Extends:</b> Root(version, description, hash, kind = "SU1Map")			
	Property	Symbol	Type	Description
	kdom	kdom	Poset	The Kleisli domain of the map.
	kcod	kcod	Poset	The Kleisli codomain of the map.
	pes	$S^{\ominus}$	Poset	The resolution poset for pessimistic estimate.
	opt	$S^{\oplus}$	Poset	The resolution poset for optimistic estimate.
	type		string	Discriminator variable to distinguish subtypes.

### Subtypes based on the value for type

"SU1_C_Parallel"	Monoidal product
"SU1_C_Series"	Series composition
"SU1_Identity"	Identity
"SU1_Unknown"	Placeholder for an unknown SU1Map
"SU1_C_CodSum"	Sum of maps
"SU1_C_CodSumSmash"	Smash sum
"SU1_C_ProdIntersection"	Product of domains, intersection of codomains
"SU1_C_Product"	Product of SU1 maps
"SU1_C_Intersection"	Intersection of SU1 maps
"SU1_C_Union"	Union of SU1 maps
"SU1_C_RefineDomain"	Refinement of the domain
"SU1_C_Trace"	Trace
"SU1_C_WrapUnits"	Wraps a map with units.
"SU1_Exact"	Lifts a U1Map to a SU1Map.
"SU1_InvMultiply"	The upper inverse of multiplication.
"SU1_InvSum"	The inverse of addition.
"SU1_C_ExplicitApprox"	Constructs a SU1Map from explicit approximations of U1Map maps.

### 26.9.1. SU1\_C\_Parallel - Monoidal product

Data	<b>Extends:</b> SU1Map(kdom, kcod, pes, opt, type = "SU1_C_Parallel")			
	Property	Symbol	Type	Description
	ms labels	$u_k$	array[SU1Map] array[string]?	Maps to be composed. A list of labels for the maps.

*Discussed in Section 22.3 (Parallel composition)*

This is the generalization of U1\_C\_Parallel.

### 26.9.2. SU1\_C\_Series - Series composition

Data	<b>Extends:</b> SU1Map(kdom, kcod, pes, opt, type = "SU1_C_Series")			
	Property	Symbol	Type	Description
	ms labels	$u_k$	array[SU1Map] array[string]?	Maps to be composed. A list of labels for the maps.

*Discussed in Section 22.4 (Series composition)*

This is the generalization of U1\_C\_Series.

### 26.9.3. **SU1\_Identity** - Identity

Data

**Extends:** `SU1Map(kdom, kcod, pes, opt, type = "SU1_Identity")`

*Discussed in Section 22.1 (Identities)*

### 26.9.4. **SU1\_Unknown** - Placeholder for an unknown SU1Map

Data

**Extends:** `SU1Map(kdom, kcod, pes, opt, type = "SU1_Unknown")`

### 26.9.5. **SU1\_C\_CodSum** - Sum of maps

Data

**Extends:** `SU1Map(kdom, kcod, pes, opt, type = "SU1_C_CodSum")`

Property	Symbol	Type	Description
<code>ms</code> labels	$u_k$	array[ <code>SU1Map</code> ] array[string]?	Maps to be composed. A list of labels for the maps.

*Discussed in Section 22.9 (Sum)*

This is the generalization of `U1_C_CodSum`.

### 26.9.6. **SU1\_C\_CodSumSmash** - Smash sum

Data

**Extends:** `SU1Map(kdom, kcod, pes, opt, type = "SU1_C_CodSumSmash")`

Property	Symbol	Type	Description
<code>ms</code> labels	$u_k$	array[ <code>SU1Map</code> ] array[string]?	Maps to be composed. A list of labels for the maps.

*Discussed in Section 22.9 (Sum)*

This is the generalization of `U1_C_CodSumSmash`.

### 26.9.7. **SU1\_C\_ProdIntersection** - Product of domains, intersection of codomains

Data

**Extends:** `SU1Map(kdom, kcod, pes, opt, type = "SU1_C_ProdIntersection")`

Property	Symbol	Type	Description
<code>ms</code> labels	$u_k$	array[ <code>SU1Map</code> ] array[string]?	Maps to be composed. A list of labels for the maps.

*Discussed in Section 22.10 (Product intersection)*

This is the generalization of `U1_C_ProdIntersection`.

### 26.9.8. **SU1\_C\_Product** - Product of SU1 maps

Data

**Extends:** `SU1Map(kdom, kcod, pes, opt, type = "SU1_C_Product")`

Property	Symbol	Type	Description
<code>ms</code> labels	$u_k$	array[ <code>SU1Map</code> ] array[string]?	Maps to be composed. A list of labels for the maps.

*Discussed in Section 22.8 (Product)*

This is the generalization of `U1_C_Product`.

### 26.9.9. **SU1\_C\_Intersection** - Intersection of SU1 maps

Data

**Extends:** `SU1Map(kdom, kcod, pes, opt, type = "SU1_C_Intersection")`

Property	Symbol	Type	Description
<code>ms</code> labels	$u_k$	array[ <code>SU1Map</code> ] array[string]?	Maps to be composed. A list of labels for the maps.

*Discussed in Section 22.6 (Intersection)*

This is the generalization of `U1_C_Intersection`.

### 26.9.10. **SU1\_C\_Union** - Union of SU1 maps

Data

**Extends:** `SU1Map(kdom, kcod, pes, opt, type = "SU1_C_Union")`

Property	Symbol	Type	Description
<code>ms</code> labels	$u_k$	array[ <code>SU1Map</code> ] array[string]?	Maps to be composed. A list of labels for the maps.

*Discussed in Section 22.5 (Union)*

This is the generalization of `U1_C_Union`.

### 26.9.11. **SU1\_C\_RefineDomain** - Refinement of the domain

Data

**Extends:** `SU1Map(kdom, kcod, pes, opt, type = "SU1_C_RefineDomain")`

Property	Symbol	Type	Description
<code>m</code>	$u$	<code>SU1Map</code>	The map to be transformed

This is the generalization of `U1_C_RefineDomain`.



### 26.9.12. **SU1\_C\_Trace** - Trace

Data

**Extends:** `SU1Map(kdom, kcod, pes, opt, type = "SU1_C_Trace")`

Property	Symbol	Type	Description
<code>m</code>	<code>u</code>	<code>SU1Map</code>	The map to be transformed

*Discussed in Section 22.7 (Trace)*

This is the generalization of `U1_C_Trace`.

### 26.9.13. **SU1\_C\_WrapUnits** - Wraps a map with units.

Data

**Extends:** `SU1Map(kdom, kcod, pes, opt, type = "SU1_C_WrapUnits")`

Property	Symbol	Type	Description
<code>m</code>	<code>u</code>	<code>SU1Map</code>	The map to be transformed
<code>kdom_units</code>		<code>Unit</code>	Units for the domain
<code>kcod_units</code>		<code>Unit</code>	Units for the codomain

This is the generalization of `U1_C_WrapUnits`.

### 26.9.14. **SU1\_Exact** - Lifts a U1Map to a SU1Map.

Data

**Extends:** `SU1Map(kdom, kcod, pes, opt, type = "SU1_Exact")`

Property	Symbol	Type	Description
<code>m</code>	<code>u</code>	<code>U1Map</code>	The U1Map to be lifted to a SU1Map.

*Discussed in Section 22.2 (Lifting)*

### 26.9.15. **SU1\_InvMultiply** - The upper inverse of multiplication.

Data

**Extends:** `SU1Map(kdom, kcod, pes, opt, type = "SU1_InvMultiply")`

Property	Symbol	Type	Description
<code>ospace</code>		<code>Poset</code>	The poset where the operation is defined.

*Discussed in Section 22.11 (Scalable inverse of sum and multiplication operations)*

### 26.9.16. **SU1\_InvSum** - The inverse of addition.

Data

**Extends:** `SU1Map(kdom, kcod, pes, opt, type = "SU1_InvSum")`

Property	Symbol	Type	Description
<code>ospace</code>		<code>Poset</code>	The poset where the operation is defined.

*Discussed in Section 22.11 (Scalable inverse of sum and multiplication operations)*

### 26.9.17. **SU1\_C\_ExplicitApprox** - Constructs a SU1Map from explicit approximations of U1Map maps.

Data

**Extends:** `SU1Map(kdom, kcod, pes, opt, type = "SU1_C_ExplicitApprox")`

Property	Symbol	Type	Description
<code>optimistic</code>		<code>array[U1Map]</code>	The optimistic approximations of the map
<code>optimistic_labels</code>		<code>array[string]?</code>	Labels for the optimistic approximations.
<code>pessimistic</code>		<code>array[U1Map]</code>	The pessimistic approximations of the map
<code>pessimistic_labels</code>		<code>array[string]?</code>	Labels for the pessimistic approximations.

*Discussed in Section 22.12 (Explicit approximation)*

## 26.10. **SLMap** - Scalable map to lower sets of functionalities and implementations.

Data

**Extends:** Root(version, description, hash, kind = "SLMap")

Property	Symbol	Type	Description
kdom	kdom	Poset	Kleisli domain of the map.
kcod	kcod	Poset	Kleisli co-domain of the map.
kimp	kimp	Poset	Poset of implementations.
pes	$S^{\ominus}$	Poset	Poset of resolutions (pessimistic)
opt	$S^{\oplus}$	Poset	Poset of resolutions (optimistic)
type		string	Discriminator variable to distinguish subtypes.

Subtypes based on the value for type

"SL_Identity"	Identity
"SL_Unknown"	Placeholder for unknown SLMap
"SL_C_Intersection"	Intersection of the results of a set of maps.
"SL_C_Parallel"	Monoidal product
"SL_C_Series"	Series composition
"SL_C_Union"	Composition of SLMaps using the union of the results.
"SL_C_ITransform"	Transforms the implementations of a SLMap.
"SL_C_RefineDomain"	Refines the domain of another SLMap
"SL_C_Trace"	Trace of a SLMap.
"SL_C_WrapUnits"	Decorates with units another SLMap.
"SL_L_Exact"	Lifts a LMap to a SLMap.
"SL_L_Explicit_Approx"	Construct a SLMap from explicit optimistic and pessimistic approximations.
"SL_L_Lift1_Constant"	Lifts a SL1Map to SLMap with a constant implementation.
"SL_L_Lift1_Transform"	Lifts a SL1Map to SLMap by generating the implementations.

### 26.10.1. **SL\_Identity** - Identity

Data

**Extends:** SLMap(kdom, kcod, kimp, pes, opt, type = "SL\_Identity")

### 26.10.2. **SL\_Unknown** - Placeholder for unknown SLMap

Data

**Extends:** SLMap(kdom, kcod, kimp, pes, opt, type = "SL\_Unknown")

### 26.10.3. **SL\_C\_Intersection** - Intersection of the results of a set of maps.

Data

**Extends:** SLMap(kdom, kcod, kimp, pes, opt, type = "SL\_C\_Intersection")

Property	Symbol	Type	Description
ms		array[SLMap]	Maps to be composed.
labels		array[string]?	A list of labels for the maps.

*Discussed in Section 23.5 (Intersection)*

This is the generalization of **L\_C\_Intersection**.

#### 26.10.4. **SL\_C\_Parallel** - Monoidal product

Data

**Extends:** `SLMap(kdom, kcod, kimp, pes, opt, type = "SL_C_Parallel")`

Property	Symbol	Type	Description
<code>ms</code>		<code>array[SLMap]</code>	Maps to be composed.
<code>labels</code>		<code>array[string]?</code>	A list of labels for the maps.

*Discussed in Section 23.3 (Parallel composition)*

This is the generalization of `L_C_Parallel`.

#### 26.10.5. **SL\_C\_Series** - Series composition

Data

**Extends:** `SLMap(kdom, kcod, kimp, pes, opt, type = "SL_C_Series")`

Property	Symbol	Type	Description
<code>ms</code>		<code>array[SLMap]</code>	Maps to be composed.
<code>labels</code>		<code>array[string]?</code>	A list of labels for the maps.

*Discussed in Section 23.4 (Series composition)*

This is the generalization of `L_C_Series`.

#### 26.10.6. **SL\_C\_Union** - Composition of SLMaps using the union of the results.

Data

**Extends:** `SLMap(kdom, kcod, kimp, pes, opt, type = "SL_C_Union")`

Property	Symbol	Type	Description
<code>ms</code>		<code>array[SLMap]</code>	Maps to be composed.
<code>labels</code>		<code>array[string]?</code>	A list of labels for the maps.

*Discussed in Section 23.6 (Union)*

This is the generalization of `L_C_Union`.

#### 26.10.7. **SL\_C\_ITransform** - Transforms the implementations of a SLMap.

Data

**Extends:** `SLMap(kdom, kcod, kimp, pes, opt, type = "SL_C_ITransform")`

Property	Symbol	Type	Description
<code>m</code>	$m$	<code>SLMap</code>	The SLMap to be transformed.
<code>transform</code>	$f$	<code>MonotoneMap</code>	The monotone map that transforms the implementations of the SLMap.

This is the generalization of `L_C_ITransform`.

### 26.10.8. **SL\_C\_RefineDomain** - Refines the domain of another SLMaP

Data

**Extends:** `SLMap(kdom, kcod, kimp, pes, opt, type = "SL_C_RefineDomain")`

Property	Symbol	Type	Description
<code>m</code>	$m$	<code>SLMap</code>	The SLMaP to be transformed.

This is the generalization of `L_C_RefineDomain`.

### 26.10.9. **SL\_C\_Trace** - Trace of a SLMaP.

Data

**Extends:** `SLMap(kdom, kcod, kimp, pes, opt, type = "SL_C_Trace")`

Property	Symbol	Type	Description
<code>m</code>	$m$	<code>SLMap</code>	The SLMaP to be transformed.
<code>m_proj</code>	$m'$	<code>SL1Map</code>	The SL1Map projection of the SLMaP $m$ .

*Discussed in Section 23.7 (Trace)*

This is the generalization of `L_C_Trace`. For computational convenience, we also require to have  $m'$ , the `SL1Map` projection of the `SLMap`  $m$ .

### 26.10.10. **SL\_C\_WrapUnits** - Decorates with units another SLMaP.

Data

**Extends:** `SLMap(kdom, kcod, kimp, pes, opt, type = "SL_C_WrapUnits")`

Property	Symbol	Type	Description
<code>m</code>	$m$	<code>SLMap</code>	The SLMaP to be transformed.
<code>kdom_units</code>		<code>Unit</code>	Units for the domain of the SLMaP.
<code>kcod_units</code>		<code>Unit</code>	Units for the codomain of the SLMaP.
<code>kimp_units</code>		<code>Unit</code>	Units for the implementations of the SLMaP.

This is the generalization of `L_C_WrapUnits`.

### 26.10.11. **SL\_L\_Exact** - Lifts a LMaP to a SLMaP.

Data

**Extends:** `SLMap(kdom, kcod, kimp, pes, opt, type = "SL_L_Exact")`

Property	Symbol	Type	Description
<code>m</code>		<code>LMaP</code>	The LMaP to be lifted to a SLMaP.

*Discussed in Section 23.1 (Lifts)*

### 26.10.12. **SL\_L\_Explicit\_Approx** - Construct a SLMaP from explicit optimistic and pessimistic approximations.

Data

**Extends:** `SLMap(kdom, kcod, kimp, pes, opt, type = "SL_L_Explicit_Approx")`

Property	Symbol	Type	Description
<code>optimistic</code>		<code>array[LMaP]</code>	The optimistic approximations of the SLMaP.
<code>optimistic_labels</code>		<code>array[string]?</code>	Labels for the optimistic approximations.
<code>pessimistic</code>		<code>array[LMaP]</code>	The pessimistic approximations of the SLMaP.
<code>pessimistic_labels</code>		<code>array[string]?</code>	Labels for the pessimistic approximations.

*Discussed in Section 23.2 (Explicit approximations)*

### 26.10.13. `SL_L_Lift1_Constant` - Lifts a `SL1Map` to `SLMap` with a constant implementation.

**Extends:** `SLMap(kdom, kcod, kimp, pes, opt, type = "SL_L_Lift1_Constant")`

Property	Symbol	Type	Description
<code>m</code> value		<code>SL1Map</code> any	The <code>SL1Map</code> to be lifted to a <code>SLMap</code> . The constant value to be used for the implementations

This is the generalization of `L_L_Lift1_Constant`.

This is the particular case of `SL_L_Lift1_Transform` where the transform map is constant.

### 26.10.14. `SL_L_Lift1_Transform` - Lifts a `SL1Map` to `SLMap` by generating the implementations.

**Extends:** `SLMap(kdom, kcod, kimp, pes, opt, type = "SL_L_Lift1_Transform")`

Property	Symbol	Type	Description
<code>m</code> <code>transform</code>	$m$ $f$	<code>SL1Map</code> <code>MonotoneMap</code>	The <code>SL1Map</code> to be lifted to a <code>SLMap</code> . The monotone map that transforms the implementations of the <code>SLMap</code> .

This is the generalization of `L_L_Lift1_Transform`.

## 26.11. SUMap - Scalable map to upper sets of resources and implementations.

Data

**Extends:** Root(version, description, hash, kind = "SUMap")

Property	Symbol	Type	Description
kdom	kdom	Poset	The Kleisli domain of the map.
kcod	kcod	Poset	The Kleisli codomain of the map.
kimp	kimp	Poset	The implementation poset of the map.
pes	$S^{\ominus}$	Poset	The resolution poset for pessimistic estimate.
opt	$S^{\oplus}$	Poset	The resolution poset for optimistic estimate.
type		string	Discriminator variable to distinguish subtypes.

Subtypes based on the value for type

"SU_Identity"	Identity
"SU_Unknown"	Placeholder for unknown SUMap
"SU_C_Intersection"	Intersection of the results of a set of maps.
"SU_C_Parallel"	Monoidal product
"SU_C_Series"	Series composition
"SU_C_Union"	Composition of SUMaps using the union of the results.
"SU_C_ITransform"	Transforms the implementations of a SUMap.
"SU_C_RefineDomain"	Refines the domain of another SUMap
"SU_C_Trace"	Trace of a SUMap.
"SU_C_WrapUnits"	Decorates with units another SUMap.
"SU_L_Exact"	Lifts a UMap to a SUMap.
"SU_L_Explicit_Approx"	Construct a SUMap from explicit optimistic and pessimistic approximations.
"SU_L_Lift1_Constant"	Lifts a SU1Map to SUMap with a constant implementation.
"SU_L_Lift1_Transform"	Lifts a SU1Map to SUMap by generating the implementations.

### 26.11.1. SU\_Identity - Identity

Data

**Extends:** SUMap(kdom, kcod, kimp, pes, opt, type = "SU\_Identity")

### 26.11.2. SU\_Unknown - Placeholder for unknown SUMap

Data

**Extends:** SUMap(kdom, kcod, kimp, pes, opt, type = "SU\_Unknown")

### 26.11.3. SU\_C\_Intersection - Intersection of the results of a set of maps.

Data

**Extends:** SUMap(kdom, kcod, kimp, pes, opt, type = "SU\_C\_Intersection")

Property	Symbol	Type	Description
ms	$m_i$	array[SUMap]	Maps to be composed
labels		array[string]?	Labels for the maps

*Discussed in Section 23.5 (Intersection)*

This is the generalization of U\_C\_Intersection.

#### 26.11.4. **SU\_C\_Parallel** - Monoidal product

Data

**Extends:** `SUMap(kdom, kcod, kimp, pes, opt, type = "SU_C_Parallel")`

Property	Symbol	Type	Description
<code>ms</code> labels	$m_i$	array[ <code>SUMap</code> ] array[string]?	Maps to be composed Labels for the maps

*Discussed in Section 23.3 (Parallel composition)*

This is the generalization of `U_C_Parallel`.

#### 26.11.5. **SU\_C\_Series** - Series composition

Data

**Extends:** `SUMap(kdom, kcod, kimp, pes, opt, type = "SU_C_Series")`

Property	Symbol	Type	Description
<code>ms</code> labels	$m_i$	array[ <code>SUMap</code> ] array[string]?	Maps to be composed Labels for the maps

*Discussed in Section 23.4 (Series composition)*

This is the generalization of `U_C_Series`.

#### 26.11.6. **SU\_C\_Union** - Composition of SUMaps using the union of the results.

Data

**Extends:** `SUMap(kdom, kcod, kimp, pes, opt, type = "SU_C_Union")`

Property	Symbol	Type	Description
<code>ms</code> labels	$m_i$	array[ <code>SUMap</code> ] array[string]?	Maps to be composed Labels for the maps

*Discussed in Section 23.6 (Union)*

This is the generalization of `U_C_Union`.

#### 26.11.7. **SU\_C\_ITransform** - Transforms the implementations of a SUMap.

Data

**Extends:** `SUMap(kdom, kcod, kimp, pes, opt, type = "SU_C_ITransform")`

Property	Symbol	Type	Description
<code>m</code> <code>transform</code>	$m$ $f$	<code>SUMap</code> <code>MonotoneMap</code>	Map to be transformed The monotone map that transforms the implementations of the SUMap.

This is the generalization of `U_C_ITransform`.



### 26.11.8. **SU\_C\_RefineDomain** - Refines the domain of another SUMap

Data

**Extends:** `SUMap(kdom, kcod, kimp, pes, opt, type = "SU_C_RefineDomain")`

Property	Symbol	Type	Description
<code>m</code>	$m$	<code>SUMap</code>	Map to be transformed

This is the generalization of `U_C_RefineDomain`. The map is refined by restricting the domain to a subset of the original domain.

### 26.11.9. **SU\_C\_Trace** - Trace of a SUMap.

Data

**Extends:** `SUMap(kdom, kcod, kimp, pes, opt, type = "SU_C_Trace")`

Property	Symbol	Type	Description
<code>m</code>	$m$	<code>SUMap</code>	Map to be transformed
<code>m_proj</code>	$m'$	<code>SU1Map</code>	The SU1Map projection of the SUMap $m$ .

*Discussed in Section 23.7 (Trace)*

This is the generalization of `U_C_Trace`.

For computational convenience, we also require to have  $m'$ , the `SU1Map` projection of the `SUMap`  $m$ .

### 26.11.10. **SU\_C\_WrapUnits** - Decorates with units another SUMap.

Data

**Extends:** `SUMap(kdom, kcod, kimp, pes, opt, type = "SU_C_WrapUnits")`

Property	Symbol	Type	Description
<code>m</code>	$m$	<code>SUMap</code>	Map to be transformed
<code>kdom_units</code>		<code>Unit</code>	Units for the domain of the SUMap.
<code>kcod_units</code>		<code>Unit</code>	Units for the codomain of the SUMap.
<code>kimp_units</code>		<code>Unit</code>	Units for the implementations of the SUMap.

This is the generalization of `U_C_WrapUnits`.

### 26.11.11. **SU\_L\_Exact** - Lifts a UMap to a SUMap.

Data

**Extends:** `SUMap(kdom, kcod, kimp, pes, opt, type = "SU_L_Exact")`

Property	Symbol	Type	Description
<code>m</code>		<code>UMap</code>	The UMap to be lifted to a SUMap.

*Discussed in Section 23.1 (Lifts)*

### 26.11.12. **SU\_L\_Explicit\_Approx** - Construct a SUMap from explicit optimistic and pessimistic approximations.

Data

**Extends:** SUMap(kdom, kcod, kimp, pes, opt, type = "SU\_L\_Explicit\_Approx")

Property	Symbol	Type	Description
optimistic		array[UMap]	The optimistic approximations of the SUMap.
optimistic_labels		array[string]?	Labels for the optimistic approximations.
pessimistic		array[UMap]	The pessimistic approximations of the SUMap.
pessimistic_labels		array[string]?	Labels for the pessimistic approximations.

*Discussed in Section 23.2 (Explicit approximations)*

### 26.11.13. **SU\_L\_Lift1\_Constant** - Lifts a SU1Map to SUMap with a constant implementation.

Data

**Extends:** SUMap(kdom, kcod, kimp, pes, opt, type = "SU\_L\_Lift1\_Constant")

Property	Symbol	Type	Description
m	m	SU1Map	The SU1Map to be lifted to a SUMap.
value		any	The constant value to be used for the implementations

This is the generalization of **U\_L\_Lift1\_Constant**.

This is the particular case of **SU\_L\_Lift1\_Transform** where the transform map is constant.

### 26.11.14. **SU\_L\_Lift1\_Transform** - Lifts a SU1Map to SUMap by generating the implementations.

Data

**Extends:** SUMap(kdom, kcod, kimp, pes, opt, type = "SU\_L\_Lift1\_Transform")

Property	Symbol	Type	Description
m	m	SU1Map	The SU1Map to be lifted to a SUMap.
transform	f	MonotoneMap	The monotone map that transforms the implementations of the SUMap.

This is the generalization of **U\_L\_Lift1\_Transform**.

## 26.12. DP - Design problem with implementations (DPI)

Data

**Extends:** Root(version, description, hash, kind = "DP")

Property	Symbol	Type	Description
<b>F</b>	<b>F</b>	Poset	Poset of functionalities
<b>R</b>	<b>R</b>	Poset	Poset of requirements
<b>B</b>	<b>B</b>	Poset?	Poset of blueprints. If not present, it is the smash unit.
<b>I</b>	<b>I</b>	Poset?	Poset of implementations. If not present, it is the smash unit.
address		Address?	Pointer to the entity that generated this object.
type		string	Discriminator variable to distinguish subtypes.

*Discussed in Section 14.1 (DPIs)*

Subtypes based on the value for type

"DP_GenericConstant"	A DP with exactly one implementation.
"DP_Identity"	The identity design problem.
"DP_True"	The DP that is always true.
"DP_False"	The DP that is always false.
"DP_AmbientConversion"	Compares functionality and resources in an ambient poset.
"DP_Catalog"	A DP defined explicitly by a set of options.
"DP_Iso"	Enforces isomorphism between functionalities and requirements.
"DP_LiftL"	A DP generated from a monotone map from requirements to functionalities.
"DP_LiftU"	A DP generated from a monotone map from functionality to requirements.
"DP_C_Parallel"	Monoidal product of design problems.
"DP_C_Series"	Series composition of DPs.
"DP_C_Intersection"	Intersection of design problems
"DP_C_Union"	Union of design problems (DPs).
"DP_C_Trace"	Trace of a design problem.
"DP_FuncNotMoreThan"	Identity with limit to the functionality.
"DP_ResNotLessThan"	Identity with limit to the resource.
"DP_All_Fi_Leq_R"	Compares a vector of functions to a resource (conjunction).
"DP_Any_Fi_Leq_R"	Compares a vector of functions to a resource (disjunction).
"DP_F_Leq_All_Ri"	Compares a vector of resources to a function (conjunction).
"DP_F_Leq_Any_Ri"	Compares a vector of resources to a function (disjunction).
"DP_All_Constants_Leq_R"	Compare a resource to a set of constants
"DP_F_Leq_All_Constants"	Compare a functionality to a set of constants
"DP_All_Constants_And_F_Leq_R"	Compares resources to a function and a set of constants (conjunction).
"DP_Any_Constants_Or_F_Leq_R"	Compares resources to a function and a set of constants (disjunction).
"DP_F_Leq_All_R_And_Constants"	Compares a functionality to a resource and a set of constants (conjunction).
"DP_F_Leq_Any_R_And_Constants"	Compares a functionality to a resource and a set of constants (disjunction).
"DP_C_ExplicitApprox"	Multi-resolution DP
"DP_Compiled"	An "opaque" DP defined explicitly by its interface.
"DP_Unknown"	Placeholder for an unknown design problem.

### 26.12.1. DP\_GenericConstant - A DP with exactly one implementation.

Data

**Extends:** DP(F, R, B, I, address, type = "DP\_GenericConstant")

Property	Symbol	Type	Description
b_value	<i>b</i>	any	The value of blueprint.
lower_set	<i>L</i>	LowerSet	The lower set of functionalities that are compatible.
upper_set	<i>U</i>	UpperSet	The upper set of resources that are compatible.

*Discussed in Section 25.2 (Catalogs)*

The relation is given by:

$$(f \in L) \quad \wedge \quad (r \in U) \quad (64)$$

26.12.2. **DP\_Identity** - The identity design problem.

Data

**Extends:** DP(F, R, B, I, address, type = "DP\_Identity")

*Discussed in Section 24.1 (Identity)*

26.12.3. **DP\_True** - The DP that is always true.

Data

<b>Extends:</b> DP(F, R, B, I, address, type = "DP_True")			
Property	Symbol	Type	Description
value		Value	The implementation value.

*Discussed in Section 25.1.1 (True)*

The relation is always true.

Examples

```
kind: DP
type: DP_True
F: {kind: Poset, type: P_C_Product, subs: []}
R: {kind: Poset, type: P_C_Product, subs: []}
I: {kind: Poset, type: P_C_ProductSmash, subs: []}
B:
  kind: Poset
  type: P_C_ProductSmash
  subs: [{kind: Poset, type: P_Decimal}]
  naked: true
  ranges: ...
value:
  kind: Value
  type: VU
  poset:
    kind: Poset
    type: P_C_ProductSmash
    subs: [{kind: Poset, type: P_Decimal}]
    naked: true
    ranges: ...
  value: [15]
```

This is a DP which is always true, whose blueprint value is  $\langle 15 \rangle$ .

26.12.4. **DP\_False** - The DP that is always false.

Data

**Extends:** DP(F, R, B, I, address, type = "DP\_False")

*Discussed in Section 25.1.2 (False)*

The relation is always false.

### 26.12.5. DP\_AmbientConversion - Compares functionality and resources in an ambient poset.

Data

**Extends:** DP(F, R, B, I, address, type = "DP\_AmbientConversion")

Property	Symbol	Type	Description
common	R	Poset	The ambient poset.

*Discussed in Section 24.2 (Ambient conversion)*

Let R be an ambient poset for both F and R:

$$\mathbf{F} \subseteq \mathbf{R}$$

$$\mathbf{R} \subseteq \mathbf{R}$$

Then this DP corresponds to the feasibility relation

$$f \leq_{\mathbf{R}} r$$

### 26.12.6. DP\_Catalog - A DP defined explicitly by a set of options.

Data

**Extends:** DP(F, R, B, I, address, type = "DP\_Catalog")

Property	Symbol	Type	Description
options		array[DP_Catalog_Options]	A list of options that define the design problem. Each option is a tuple of functionality, requirement, blueprint, and implementation.

*Discussed in Section 25.2 (Catalogs)*

This DP is defined explicitly by a set of options. Each option is a tuple of functionality, requirement, blueprint, and implementation.

The relation is given by:

$$\bigvee_{(f_j, r_j, b_j, i_j) \in \text{options}} (f \leq f_j) \wedge (r_j \leq r)$$

#### DP\_Catalog\_Options - One option for the catalog

Data

Property	Symbol	Type	Description
f		any	Functionality
r		any	Requirement
b		any	Blueprint
i		any	Implementation

### 26.12.7. **DP\_Iso** - Enforces isomorphism between functionalities and requirements.

Data	<b>Extends:</b> DP( <b>F</b> , <b>R</b> , <b>B</b> , <b>I</b> , address, type = "DP_Iso")			
	Property	Symbol	Type	Description
	<b>fwd</b>	$\alpha$	MonotoneMap	A monotone map from the poset of functionalities to the poset of requirements.
	<b>bwd</b>	$\beta$	MonotoneMap	A monotone map from the poset of requirements to the poset of functionalities.

*Discussed in Section 24.3 (Isomorphism)*

The isomorphism is defined by the two monotone maps  $\alpha : \mathbf{F} \rightarrow \mathbf{R}$  and  $\beta : \mathbf{R} \rightarrow \mathbf{F}$  such that  $\alpha \circ \beta = \text{id}$  and  $\beta \circ \alpha = \text{id}$ .

### 26.12.8. **DP\_LiftL** - A DP generated from a monotone map from requirements to functionalities.

Data	<b>Extends:</b> DP( <b>F</b> , <b>R</b> , <b>B</b> , <b>I</b> , address, type = "DP_LiftL")			
	Property	Symbol	Type	Description
	<b>m</b>	$m$	MonotoneMap	A monotone map from the poset of requirements to the poset of functionalities.

The relation is

$$f \leq m(r) \quad (65)$$

### 26.12.9. **DP\_LiftU** - A DP generated from a monotone map from functionality to requirements.

Data	<b>Extends:</b> DP( <b>F</b> , <b>R</b> , <b>B</b> , <b>I</b> , address, type = "DP_LiftU")			
	Property	Symbol	Type	Description
	<b>m</b>	$m$	MonotoneMap	A monotone map from the poset of functionalities to the poset of requirements. This is used to lift a DP from the requirements to the functionalities.

*Discussed in Section 24.5 (Upper lift of a map)*

The relation is

$$m(f) \leq r \quad (66)$$

### 26.12.10. **DP\_C\_Parallel** - Monoidal product of design problems.

Data	<b>Extends:</b> DP( <b>F</b> , <b>R</b> , <b>B</b> , <b>I</b> , address, type = "DP_C_Parallel")			
	Property	Symbol	Type	Description
	<b>dps</b> labels		array[DP] array[string]?	A list of design problems (DPs) to be composed. A list of labels.

*Discussed in Section 25.3 (Parallel composition)*

#### 26.12.11. DP\_C\_Series - Series composition of DPs.

Data

**Extends:** DP(F, R, B, I, address, type = "DP\_C\_Series")

Property	Symbol	Type	Description
dps		array[DP]	A list of design problems (DPs) to be composed.
labels		array[string]?	A list of labels.

*Discussed in Section 25.4 (Series)*

Series composition of DPs.

#### 26.12.12. DP\_C\_Intersection - Intersection of design problems

Data

**Extends:** DP(F, R, B, I, address, type = "DP\_C\_Intersection")

Property	Symbol	Type	Description
dps		array[DP]	A list of design problems (DPs) to be composed.
labels		array[string]?	A list of labels.

*Discussed in Section 25.5 (Intersection)*

#### 26.12.13. DP\_C\_Union - Union of design problems (DPs).

Data

**Extends:** DP(F, R, B, I, address, type = "DP\_C\_Union")

Property	Symbol	Type	Description
dps		array[DP]	A list of design problems (DPs) to be composed.
labels		array[string]?	A list of labels.

*Discussed in Section 25.6 (Union)*

#### 26.12.14. DP\_C\_Trace - Trace of a design problem.

Data

**Extends:** DP(F, R, B, I, address, type = "DP\_C\_Trace")

Property	Symbol	Type	Description
dp		DP	The design problem that is being traced.

*Discussed in Section 25.7 (Trace)*

#### 26.12.15. DP\_FuncNotMoreThan - Identity with limit to the functionality.

Data

**Extends:** DP(F, R, B, I, address, type = "DP\_FuncNotMoreThan")

Property	Symbol	Type	Description
limit	$L$	any	The limit for the functionality.

*Discussed in* Section 24.6.1 (Functionality not more than the requirement and constant)

The relation is given by:

$$(f \leq L) \wedge (f \leq r)$$

#### 26.12.16. DP\_ResNotLessThan - Identity with limit to the resource.

Data

**Extends:** DP(F, R, B, I, address, type = "DP\_ResNotLessThan")

Property	Symbol	Type	Description
limit	$L$	any	The limit for the resource.

*Discussed in* Section 24.6.2 (Requirement not less than the functionality and constant)

The relation is given by:

$$(f \leq r) \wedge (L \leq r)$$

#### 26.12.17. DP\_All\_Fi\_Leq\_R - Compares a vector of functions to a resource (conjunction).

Data

**Extends:** DP(F, R, B, I, address, type = "DP\_All\_Fi\_Leq\_R")

*Discussed in* Section 24.6.3 (All functionalities less than the requirement)

The relation is given by:

$$(f_1 \leq r) \wedge \dots \wedge (f_n \leq r)$$

#### 26.12.18. DP\_Any\_Fi\_Leq\_R - Compares a vector of functions to a resource (disjunction).

Data

**Extends:** DP(F, R, B, I, address, type = "DP\_Any\_Fi\_Leq\_R")

*Discussed in* Section 24.6.4 (Any functionality less than the requirement)

The relation is given by:

$$(f_1 \leq r) \vee \dots \vee (f_n \leq r)$$



### 26.12.19. DP\_F\_Leq\_All\_Ri - Compares a vector of resources to a function (conjunction).

Data

**Extends:** DP(F, R, B, I, address, type = "DP\_F\_Leq\_All\_Ri")

*Discussed in Section 24.6.5 (All requirements more than the functionality)*

The relation is given by:

$$(f \leq r_1) \wedge \dots \wedge (f \leq r_n)$$

Compare with DP\_F\_Leq\_Any\_Ri which uses disjunctions instead of conjunctions.

### 26.12.20. DP\_F\_Leq\_Any\_Ri - Compares a vector of resources to a function (disjunction).

Data

**Extends:** DP(F, R, B, I, address, type = "DP\_F\_Leq\_Any\_Ri")

*Discussed in Section 24.6.6 (Any requirement more than the functionality)*

The relation is given by:

$$(f \leq r_1) \vee \dots \vee (f \leq r_n)$$

Compare with DP\_F\_Leq\_All\_Ri which uses conjunctions instead of disjunctions.

### 26.12.21. DP\_All\_Constants\_Leq\_R - Compare a resource to a set of constants

Data

**Extends:** DP(F, R, B, I, address, type = "DP\_All\_Constants\_Leq\_R")

Property	Symbol	Type	Description
constants	$c_i$	array[any]	A list of constants $c_1, \dots, c_n$ .

*Discussed in Section 24.6.7 (All constants less than the requirement)*

The relation is given by:

$$(c_1 \leq r) \wedge \dots \wedge (c_n \leq r)$$

Note: This extended form is needed when the constants are not a lattice. In the case of a lattice, the relation would be given by:

$$\bigvee_i c_i \leq r$$

and could be realized using DP\_GenericConstant.

### 26.12.22. DP\_F\_Leq\_All\_Constants - Compare a functionality to a set of constants

Data

**Extends:** DP(F, R, B, I, address, type = "DP\_F\_Leq\_All\_Constants")

Property	Symbol	Type	Description
constants	$c_i$	array[any]	A list of constants $c_1, \dots, c_n$ .

*Discussed in Section 24.6.8 (Functionality less than all constants)*

The relation is given by:

$$(f \leq c_1) \wedge \dots \wedge (f \leq c_n)$$

Note: This extended form is needed when the constants are not a lattice. In the case of a lattice, the relation would be given by:

$$f \leq \bigwedge_i c_i$$

and could be realized using DP\_GenericConstant.

### 26.12.23. DP\_All\_Constants\_And\_F\_Leq\_R - Compares resources to a function and a set of constants (conjunction).

Data

**Extends:** DP(F, R, B, I, address, type = "DP\_All\_Constants\_And\_F\_Leq\_R")

Property	Symbol	Type	Description
constants	$c_i$	array[any]	A list of constants $c_1, \dots, c_n$ .

*Discussed in Section 24.6.9 (Functionality and all constants less than the requirement)*

The relation is given by:

$$(c_1 \leq r) \wedge \dots \wedge (c_n \leq r) \wedge (f \leq r)$$

### 26.12.24. DP\_Any\_Constants\_Or\_F\_Leq\_R - Compares resources to a function and a set of constants (disjunction).

Data

**Extends:** DP(F, R, B, I, address, type = "DP\_Any\_Constants\_Or\_F\_Leq\_R")

Property	Symbol	Type	Description
constants	$c_i$	array[any]	A list of constants $c_1, \dots, c_n$ .

*Discussed in Section 24.6.10 (Functionality or any constant less than the requirement)*

The relation is given by:

$$(c_1 \leq r) \vee \dots \vee (c_n \leq r) \vee (f \leq r)$$

**26.12.25. DP\_F\_Leq\_All\_R\_And\_Constants** - Compares a functionality to a resource and a set of constants (conjunction).

Data

**Extends:** DP(F, R, B, I, address, type = "DP\_F\_Leq\_All\_R\_And\_Constants")

Property	Symbol	Type	Description
constants	$c_i$	array[any]	A list of constants $c_1, \dots, c_n$ .

*Discussed in* Section 24.6.11 (Functionality less than the requirement and all constants)

The relation is given by:

$$(f \leq c_1) \wedge \dots \wedge (f \leq c_n) \wedge (f \leq r)$$

**26.12.26. DP\_F\_Leq\_Any\_R\_And\_Constants** - Compares a functionality to a resource and a set of constants (disjunction).

Data

**Extends:** DP(F, R, B, I, address, type = "DP\_F\_Leq\_Any\_R\_And\_Constants")

Property	Symbol	Type	Description
constants	$c_i$	array[any]	A list of constants $c_1, \dots, c_n$ .

*Discussed in* Section 24.6.12 (Functionality less than the requirement or any constant)

The relation is given by:

$$(f \leq c_1) \vee \dots \vee (f \leq c_n) \vee (f \leq r)$$

### 26.12.27. DP\_C\_ExplicitApprox - Multi-resolution DP

Data

**Extends:** DP(F, R, B, I, address, type = "DP\_C\_ExplicitApprox")

Property	Symbol	Type	Description
<b>optimistic</b>		array[DP]	List of optimistic DPs.
optimistic_labels		array[string]?	Labels for the optimistic DPs.
<b>pessimistic</b>		array[DP]	List of pessimistic DPs.
pessimistic_labels		array[string]?	Labels for the pessimistic DPs.

This DP assembles a "multi resolution" DP from a list of optimistic and pessimistic DPs.

Examples

```

kind: DP
type: DP_C_ExplicitApprox
F: {kind: Poset, type: P_Decimal}
R: {kind: Poset, type: P_Decimal}
optimistic:
- kind: DP
  type: DP_True
  F: {kind: Poset, type: P_Decimal}
  R: {kind: Poset, type: P_Decimal}
  value:
    kind: Value
    type: VU
    value: []
    poset:
      kind: Poset
      type: P_C_ProductSmash
      subs: []
      naked: []
      ranges: []
pessimistic:
- kind: DP
  type: DP_False
  F: {kind: Poset, type: P_Decimal}
  R: {kind: Poset, type: P_Decimal}

```

### 26.12.28. DP\_Compiled - An "opaque" DP defined explicitly by its interface.

Data

**Extends:** DP(F, R, B, I, address, type = "DP\_Compiled")

Property	Symbol	Type	Description
<b>f_r</b>		SU1Map	The function that returns minimal resources needed to satisfy the requirements.
<b>f_b_r</b>		SUMap	The function that returns the maximum functionality given the budget of resources as well as the blueprint.
<b>f_i_r</b>		SUMap	The function that returns the maximum functionality given the budget of resources as well as the implementation.
<b>r_f</b>		SL1Map	The function that returns the maximum functionality given the budget of requirements.
<b>r_b_f</b>		SLMap	The function that returns the maximum functionality given the budget of resources as well as the blueprint.
<b>r_i_f</b>		SLMap	The function that returns the maximum functionality given the budget of resources as well as the implementation.
<b>i_b</b>		MonotoneMap	The function that maps implementations to blueprints.
<b>i_codfeas</b>		MonotoneMap	The function that maps implementations to their internal feasibility.
<b>i_availability</b>		MonotoneMap	The function that maps implementations to their availability.
<b>prov</b>		MonotoneMap	The "provides" map from implementations to functionalities.
<b>req</b>		MonotoneMap	The "requires" map from implementations to requirements.

### 26.12.29. DP\_Unknown - Placeholder for an unknown design problem.

Data

**Extends:** DP(F, R, B, I, address, type = "DP\_Unknown")

## 26.13. NDP - Named DPs represent a graph of DPs with named nodes and node ports.

Data	<b>Extends:</b> Root(version, description, hash, kind = "NDP")			
	<b>Property</b>	<b>Symbol</b>	<b>Type</b>	<b>Description</b>
	F		dict[string,Poset]	Dictionary of functionalities.
	R		dict[string,Poset]	Dictionary of resources.
	address		Address?	Pointer to the entity that generated this object.
	type		string	Discriminator variable to distinguish subtypes.

Subtypes based on the value for type

"NDP_Composite"	Graph of NDPs with connections between them.
"NDP_Simple"	An NDP that contains a single DP.
"NDP_Sum"	sum of NDPs
"NDP_TemplateHole"	A special NDP to indicate a template hole in the NDP.

### 26.13.1. NDP\_Composite - Graph of NDPs with connections between them.

Data	<b>Extends:</b> NDP(F, R, address, type = "NDP_Composite")			
	<b>Property</b>	<b>Symbol</b>	<b>Type</b>	<b>Description</b>
	nodes		dict[string,NDP]	A map of node identifiers to their corresponding NDPs in the graph. Each key is a unique identifier for a node, and the value is the NDP associated with that node.
	connections		array[Connection]	Connections between the nodes in the NDP graph.

#### Connection - Represents a connection between two nodes in the NDP graph

Data	<b>Property</b>	<b>Symbol</b>	<b>Type</b>	<b>Description</b>
	type		string	Type marker.
	source		ConnectionSource	<i>Must be equal to "Connection"</i> The source of the connection.
	target		ConnectionTarget	The target of the connection.

#### ConnectionTarget - The target of a connection.

Data	<b>Property</b>	<b>Symbol</b>	<b>Type</b>	<b>Description</b>
	type		string	Discriminator variable to distinguish subtypes.

Subtypes based on the value for type

"ModelRequirement"	The target is the requirement of the ambient model.
"NodeFunctionality"	The target is the functionality of another subproblem.

#### ModelRequirement - The target is the requirement of the ambient model.

Data	<b>Extends:</b> ConnectionTarget(type = "ModelRequirement")			
	<b>Property</b>	<b>Symbol</b>	<b>Type</b>	<b>Description</b>
	requirement		string	The name of a requirement of the ambient model.

**NodeFunctionality** - The target is the functionality of another subproblem.

Data

**Extends:** ConnectionTarget(type = "NodeFunctionality")

Property	Symbol	Type	Description
node		string	The name of the node that provides the functionality.
node_functionality		string	The name of the functionality that is provided by the node.

**ConnectionSource** - The source of a connection.

Data

Property	Symbol	Type	Description
type		string	Discriminator variable to distinguish subtypes.

Subtypes based on the value for type

"ModelFunctionality"	The source of a connection is a functionality of the composite graph.
"NodeRequirement"	The source of a connection is a requirement of another node.

**ModelFunctionality** - The source of a connection is a functionality of the composite graph.

Data

**Extends:** ConnectionSource(type = "ModelFunctionality")

Property	Symbol	Type	Description
functionality		string	The name of the functionality that is provided by the composite graph.

**NodeRequirement** - The source of a connection is a requirement of another node.

Data

**Extends:** ConnectionSource(type = "NodeRequirement")

Property	Symbol	Type	Description
node		string	The name of the node that provides the requirement.
node_requirement		string	The name of the requirement that is provided by the node.

**26.13.2. NDP\_Simple** - An NDP that contains a single DP.

Data

**Extends:** NDP(F, R, address, type = "NDP\_Simple")

Property	Symbol	Type	Description
dp		DP	The DP that this NDP contains. Must have poset products as resources and functionalities.

**26.13.3. NDP\_Sum** - sum of NDPs

Data

**Extends:** NDP(F, R, address, type = "NDP\_Sum")

Property	Symbol	Type	Description
dps		dict[string,NDP]	The NDPs to sum.
labels		array[string]?	Labels for the NDPs.

#### 26.13.4. NDP\_TemplateHole - A special NDP to indicate a template hole in the NDP.

Data

**Extends:** NDP(F, R, address, type = "NDP\_TemplateHole")

Property	Symbol	Type	Description
parameter_name		string	The name of the parameter that is to be filled in.

26.14. **NDPInterface** - The interface of a named DP.

<b>Extends:</b> Root(version, description, hash, kind = "NDPInterface")			
Property	Symbol	Type	Description
address		Address?	Pointer to the entity that generated this object.
type		string	Discriminator variable to distinguish subtypes.
Subtypes based on the value for type			
"NDPInterface_Explicit"		The interface of a named DP, given by two dictionaries for functionalities and resources.	

26.14.1. **NDPInterface\_Explicit** - The interface of a named DP, given by two dictionaries for functionalities and resources.

<b>Extends:</b> NDPInterface(address, type = "NDPInterface_Explicit")			
Property	Symbol	Type	Description
fs		dict[string,Poset]	Dictionary from functionality name to poset.
rs		dict[string,Poset]	Dictionary from requirement name to poset.



## 26.15. NDPTemplate - A template for an NDP.

Data

**Extends:** Root(version, description, hash, kind = "NDPTemplate")

Property	Symbol	Type	Description
address		Address?	Pointer to the entity that generated this object.
type		string	Discriminator variable to distinguish subtypes.

Subtypes based on the value for type

"NDPTemplate\_Simple" | A template described by a graph with holes.

### 26.15.1. NDPTemplate\_Simple - A template described by a graph with holes.

Data

**Extends:** NDPTemplate(address, type = "NDPTemplate\_Simple")

Property	Symbol	Type	Description
ndp	$N$	NDP	The NDP that this template can instantiate. Inside, there are special "holes".
parameters		dict[string,NDPInterface]	The interface of the holes.

## 26.16. Query - Queries

Data

**Extends:** Root(version, description, hash, kind = "Query")

Property	Symbol	Type	Description
address type		Address? string	Pointer to the entity that generated this object. Discriminator variable to distinguish subtypes.

Queries represent the questions that can be asked about a model.

Subtypes based on the value for type

"Query_Single"		Single query
----------------	--	--------------

### 26.16.1. Query\_Single - Single query

Data

**Extends:** Query(address, type = "Query\_Single")

Property	Symbol	Type	Description
model query_data		NDP QueryData	The model to which the query applies. The data that is used to answer the query.

#### QueryData - Query data

Data

Property	Symbol	Type	Description
type		string	Discriminator variable to distinguish subtypes.

Subtypes based on the value for type

"QueryFixFunMinReqData"		Data for the query FixFunMinReq
"QueryFixReqMaxFunData"		Data for the query FixReqMaxFun

#### QueryFixFunMinReqData - Data for the query FixFunMinReq

Data

**Extends:** QueryData(type = "QueryFixFunMinReqData")

Property	Symbol	Type	Description
f r optimize_for		dict[string,Value] dict[string,Value] array[string]	Lower bounds for the functionalities. Upper bounds for the requirements. The names of the functionalities that are optimized for. This is a list of the keys of the 'f' object.

#### QueryFixReqMaxFunData - Data for the query FixReqMaxFun

Data

**Extends:** QueryData(type = "QueryFixReqMaxFunData")

Property	Symbol	Type	Description
f r optimize_for		dict[string,Value] dict[string,Value] array[string]	Lower bounds for the functionalities. Upper bounds for the requirements. The functionalities that are optimized for. This is a list of the keys of the 'f' object.

26.17. Value - A typed value

Extends: Root(version, description, hash, kind = "Value")			
Property	Symbol	Type	Description
address type		Address? string	Pointer to the entity that generated this object. Discriminator variable to distinguish subtypes.

A typed value is a value that has a type (poset). This is used in contexts where the poset to which the value belongs is not clear from the context.

Subtypes based on the value for type	
"VU"	A (poset, value) pair.

26.17.1. VU - A (poset, value) pair.

Extends: Value(address, type = "VU")			
Property	Symbol	Type	Description
poset value	$P$ $v \in P$	Poset any	The poset to which the value belongs. The value that belongs to the poset.

## 26.18. Check - Checks for the maps, as used in test cases.

Data	<b>Extends:</b> Root(version, description, hash, kind = "Check")			
	<b>Property</b>	<b>Symbol</b>	<b>Type</b>	<b>Description</b>
	type		string	Discriminator variable to distinguish subtypes.
Subtypes based on the value for type				
	"L1Check"		Check for a L1Map.	
	"LCheck"		Check for a LMap.	
	"MapCheck"		Check for a monotone map.	
	"SL1Check"		Check for a SL1Map.	
	"SLCheck"		Check for a SL1Map.	
	"SU1Check"		Check for a SU1Map.	
	"SUCheck"		Check for a SUMap.	
	"U1Check"		Check for a U1Map.	
	"UCheck"		Check for a UMap.	

### 26.18.1. L1Check - Check for a L1Map.

Data	<b>Extends:</b> Check(type = "L1Check")			
	<b>Property</b>	<b>Symbol</b>	<b>Type</b>	<b>Description</b>
	<i>m</i> data	<i>m</i>	L1Map array[L1Check_Data]	The map to check Data to check the L1Map.

L1Check\_Data - An input-output pair for the L1Map

Data	<b>Property</b>	<b>Symbol</b>	<b>Type</b>	<b>Description</b>
	x y elapsed	<i>x</i>	any LowerSet number?	The input to the L1Map. Expected result (a lower set). Time taken for the check in seconds (optional).

### 26.18.2. LCheck - Check for a LMap.

Data	<b>Extends:</b> Check(type = "LCheck")			
	<b>Property</b>	<b>Symbol</b>	<b>Type</b>	<b>Description</b>
	<i>m</i> data	<i>m</i>	LMap array[LCheck_Data]	The map to check Test pairs

LCheck\_Data - An input-output pair for the map

Data	<b>Property</b>	<b>Symbol</b>	<b>Type</b>	<b>Description</b>
	x y elapsed	<i>x</i> <i>y</i>	any LowerSet number?	The input to the map The expected result (a lower set). Time taken for the check in seconds.

### 26.18.3. MapCheck - Check for a monotone map.

Data	Extends: Check(type = "MapCheck")			
	Property	Symbol	Type	Description
	<i>m</i> data	<i>m</i>	MonotoneMap array[MapCheck_Data]	The map to check. An input-output pair, where 'x' is the input to the map and 'y' is the expected result.

MapCheck\_Data - An input-output pair for the map

Data	Property	Symbol	Type	Description
	x y elapsed	<i>x</i> <i>y</i>	any any number?	The input to the map. The expected result Time taken to compute the result.

### 26.18.4. SL1Check - Check for a SL1Map.

Data	Extends: Check(type = "SL1Check")			
	Property	Symbol	Type	Description
	<i>m</i> data	<i>m</i>	SL1Map array[SL1Check_Data]	The map to check Test pairs

SL1Check\_Data - An input-output pair

Data	Property	Symbol	Type	Description
	x pess opt pess_y opt_y pess_elapsed opt_elapsed		any any any LowerSet LowerSet number? number?	Time taken for the check in seconds. Time taken for the check in seconds.

### 26.18.5. SLCheck - Check for a SL1Map.

Data	Extends: Check(type = "SLCheck")			
	Property	Symbol	Type	Description
	<i>m</i> data	<i>m</i>	SLMap array[SLCheck_Data]	The map to check Test pairs

SLCheck\_Data - An input-output pair for the SLMap

Data	Property	Symbol	Type	Description
	x pess opt pess_y opt_y		any any any LowerSet LowerSet	

Data	number?	Time taken for the check in seconds.
peess_elapsed	number?	Time taken for the check in seconds.
opt_elapsed	number?	Time taken for the check in seconds.

### 26.18.6. SU1Check - Check for a SU1Map.

Data	<b>Extends:</b> Check(type = "SU1Check")			
	<b>Property</b>	<b>Symbol</b>	<b>Type</b>	<b>Description</b>
	<i>m</i> data	<i>m</i>	SU1Map array[SU1Check_Data]	The map to check Test pairs

**SU1Check\_Data** - An input-output pair for the SU1Map

Data	Property	Symbol	Type	Description
	x		any	
	pass		any	
	opt		any	
	pass_y		UpperSet	
	opt_y		UpperSet	
	pass_elapsed		number?	Time taken for the check in seconds.
	opt_elapsed		number?	Time taken for the check in seconds.

### 26.18.7. SUCheck - Check for a SUMap.

Extends: Check(type = "SUCheck")			
Property	Symbol	Type	Description
<i>m</i> data	<i>m</i>	SUMap array[SUCheck_Data]	The map to check Test pairs

**SUCheck\_Data** - An input-output pair for the SUMap

Property	Symbol	Type	Description
x		any	
opt		any	
opt_y		UpperSet	
press		any	
press_y		UpperSet	
press_elapsed		number?	Time taken for the check in seconds.
opt_elapsed		number?	Time taken for the check in seconds.

### 26.18.8. U1Check - Check for a U1Map.

Data	<b>Extends:</b> Check(type = "U1Check")			
	<b>Property</b>	<b>Symbol</b>	<b>Type</b>	<b>Description</b>
	<b>m</b> data	<i>m</i>	<b>U1Map</b> array[U1Check_Data]	The map to check Test pairs

#### U1Check\_Data - An input-output pair for the U1Map

Data

Property	Symbol	Type	Description
x y elapsed		any UpperSet number?	Time taken for the check in seconds.

#### 26.18.9. UCheck - Check for a UMap.

Data

**Extends:** Check(type = "UCheck")

Property	Symbol	Type	Description
<i>m</i> data	<i>m</i>	UMap array[UCheck_Data]	The map to check Test pairs

#### UCheck\_Data - An input-output pair for the UMap

Data

Property	Symbol	Type	Description
x y elapsed		any UpperSet number?	Time taken for the check in seconds.

## 27. Miscellaneous level



## 27.1. LowerSet - Represents a lower set in a poset.

Data

Property	Symbol	Type	Description
kind		string	Kind marker. <i>Must be equal to "LowerSet"</i>
type		string	Discriminator variable to distinguish subtypes.

Subtypes based on the value for type

"LowerSet_LowerClosure"		A lower set defined as the down closure of a finite set of points.
-------------------------	--	--

### 27.1.1. LowerSet\_LowerClosure - A lower set defined as the down closure of a finite set of points.

Data

**Extends:** LowerSet(kind, type = "LowerSet\_LowerClosure")

Property	Symbol	Type	Description
points		array[any]	The points in the lower set.

## 27.2. Range - Description of a range of integers.

Data

Property	Symbol	Type	Description
start		integer	Start of the range (inclusive).
stop		integer	End of the range (exclusive).
ntot		integer	Total number of elements in the array.
type		string	Type marker <i>Must be equal to "Range"</i>

This represents the range `start:stop` (inclusive of start, exclusive of stop) in a total of `ntot` elements.

## 27.3. Unit - Units specifications

Data

Property	Symbol	Type	Description
kind		string	Kind marker. <i>Must be equal to "Unit"</i>
description		string?	A human-readable description of the unit (debug purposes).
type		string	Discriminator variable to distinguish subtypes.

Subtypes based on the value for type

"Unit_None"	Represents the absence of units.
"Unit_Single"	A simple unit.
"Unit_Vector"	A vector of units for a product of posets.
"Unit_Wrapped"	A special type of unit that is used to describe the units of composite types.

### 27.3.1. Unit\_None - Represents the absence of units.

Data

**Extends:** Unit(kind, description, type = "Unit\_None")

### 27.3.2. Unit\_Single - A simple unit.

Data

**Extends:** Unit(kind, description, type = "Unit\_Single")

Property	Symbol	Type	Description	Example
units		string	A string representing the unit.	m

### 27.3.3. Unit\_Vector - A vector of units for a product of posets.

Data

**Extends:** Unit(kind, description, type = "Unit\_Vector")

Property	Symbol	Type	Description
subs		array[Unit]	The subunits.
labels		array[string]?	labels for the subunits

### 27.3.4. Unit\_Wrapped - A special type of unit that is used to describe the units of composite types.

Data

**Extends:** Unit(kind, description, type = "Unit\_Wrapped")

Property	Symbol	Type	Description
name		string	
inside		array[Unit]	
shape		any	

## 27.4. UpperSet - Upper sets

Data

Property	Symbol	Type	Description
kind		string	Kind marker. <i>Must be equal to "UpperSet"</i>
type		string	Discriminator variable to distinguish subtypes.

Subtypes based on the value for type

"UpperSet_UpperClosure"		An upper set defined as the up closure of a finite set of points.
-------------------------	--	---

### 27.4.1. UpperSet\_UpperClosure - An upper set defined as the up closure of a finite set of points.

Data

**Extends:** UpperSet(kind, type = "UpperSet\_UpperClosure")

Property	Symbol	Type	Description
points		array[any]	The points in the set.

### 27.5. Address - Specifies the origin of an object from a repo and a library.

Property	Symbol	Type	Description
type		string	Type marker <i>Must be equal to "Address"</i>
repo		string?	The Git repository URL
library		string	The library name
spec		string	The type of object <i>Possible values: "models", "templates", "values", "posets", "primitivedps", "interfaces", "queries"</i>
thing		string	The name of the object