Quantum Kan Extensions

\[ \text{Set} \xrightarrow{F} \text{Lan}_F(X) \]

--- Final Technical Report ---

Prepared by

Baker Mountain Research Corporation
P. O. Box 68
Yellow Spring, WV 26865-0068

Dr. Ralph L. Wojtowicz

Brooklyn College
2900 Bedford Avenue
Brooklyn, NY 11210

Dr. Noson S. Yanofsky

Contract: D11PC20232

15 January 2012

Distribution Notice: Supported by the Intelligence Advanced Research Projects Activity (IARPA) via Department of Interior National Business Center contract number D11PC20232. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright annotation thereon. Disclaimer: The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of IARPA, DoI/NBC, or the U.S. Government.
This page was intentionally left blank.
Executive Summary

The objective of this work is to increase our understanding of the tasks that quantum computers will and will not be able to perform more efficiently than classical computers. Our approach is to research quantum algorithms for computing mathematical constructions known as Kan extensions.

Conclusions: Figure 1 illustrates connections between algorithms that we researched. Those in green are known quantum algorithms. We showed that the constructions in blue cannot be calculated more efficiently on a quantum computer than on a classical computer. An arrow in the figure from node $A$ to node $B$ indicates that an algorithm for computing $A$ provides an algorithm for computing $B$.

- The Hidden Subgroup Problem can be expressed as a Kan extension. Quantum algorithms for efficiently solving such problems, therefore, efficiently compute particular Kan extensions.
- Quantum computers, however, cannot provide superior methods for exact Kan extension calculations in general.
- The classical coequalizer algorithm is a special case of the Dominating Set Problem, which is NP-complete. Improved quantum bounds for coequalizers and related problems are, therefore, of value.
- Known quantum algorithms that promise an exponential speedup are formulated in a context in which a rich mathematical structure, such as that of an abelian group, is available but which cannot be exploited for general Kan extensions. This suggests new domains for seeking quantum algorithms.
- Discovery of new quantum circuits is an essential stage in understanding the benefits of quantum computers and features of quantum compilers. Availability of quantum programming languages designed with these insights supports formulation of quantum algorithms.

![Figure 1: Quantum algorithms in the context of Kan extensions](image-url)
Research Questions:

1. The potential of quantum computers to solve hidden subgroup problems far faster than classical computers was established using the quantum Fourier transform which relies on the availability of suitable underlying mathematical structures. To find novel quantum algorithms, it may be useful to focus on areas in which different, rich mathematical structure is available. Examples include order-theoretic structures such as lattices, Heyting algebras and locales. High-complexity algorithms of interest include inference in fragments of the predicate calculus and other logics.

2. Determine (1) tight bounds on the quantum complexity of coequalizer calculations (2) research other special cases of the Dominating Set Problem.

3. Research development of a compiler that supports expression of algorithms at a high conceptual level (e.g., in a higher-level programming language) but which has the capacity to generate quantum circuits that have minimal quantum resource requirements.

Peter Shor [118] discovered that a quantum computer can factor integers exponentially faster than the best known classical algorithms. Other tasks for which quantum computers were subsequently shown to provide an exponential speedup over their classical counterparts, are related to Shor’s algorithm through the concept of the Hidden Subgroup Problem and employ the quantum Fourier transform. Lov Grover [73] discovered a quantum search algorithm that is quadratically faster than the best classical procedure. The underlying technique of amplitude amplification establishes related tasks for which quantum computers promise a polynomial speedup.

The question that arises, however, is: What other tasks will quantum computers be able to perform better than classical machines? We researched answers to this question in the context of a general class of mathematical tasks called Kan extensions. Mathematical constructions which are particular examples of Kan extensions include: addition and multiplication of integers, unions and cartesian products of sets, equivalence relations and solution sets of equations. Algorithms to calculate special cases of Kan extensions include: integer arithmetic circuits, Union-Find and Quick-Union for solving graph connectivity problems and the Todd-Coxeter Procedure for enumerating cosets of a subgroup.

Chapter Summaries: In Chapter 1 of this report we give an introduction to our research approach and the problems we have worked to solve.

Chapter 2 of this report includes definitions and theorems from category theory. Right and left Kan extensions are defined in 2.81 and 2.82 respectively. 2.813 demonstrates that any right Kan extension may be computed as limit of functor constructed from the ingredients used to specify it. From this and 2.564 it follows that all right Kan extensions can be computed from two particular kinds of limits: products and equalizers. Similarly, 2.824 demonstrates that any left Kan extension may be computed as colimit while 2.661 computes arbitrary colimits from two particular kinds: coproducts and coequalizers.

The Kan extensions for which we seek quantum algorithms involve functors $X : \mathcal{A} \to \textbf{Set}_f$ where $\mathcal{A}$ is a finitely-presented category and $\textbf{Set}_f$ has finite sets as objects. In Chapter 3 of this report we detail the category $\textbf{Set}$ of sets and functions. In particular, we review formulas for computing particular cases of Kan extensions (i.e., limits and colimits) in $\textbf{Set}$. We review the equivalence between the categories $\textbf{Set}_f$ of finite sets and $\Delta$ of simplicial sets. This equivalence reduces the apparent complexity of the algorithms under investigation. The categorical product, for example, is reduced to multiplication of natural numbers and the coproduct is reduced to addition.
Chapter 4 of this report we discuss group theory. In this chapter we review the notion of a presentation of a group in terms of generators and relations, group actions, orbits, stabilizers and other concepts as they arise in the context of our work. In 4.2 we discuss the Todd-Coxeter Procedure for enumerating cosets of a subgroup. The Carmody-Walters Algorithm for computing Kan extensions is a generalization of this procedure. 4.22 and 4.23 discuss computational complexity of the Todd-Coxeter Procedure and its relation to the Word Problem. In 4.3, we discuss the Abelian Stabilizer Problem. Given the stabilizer of an element of a group action, we characterize the available mappings as a left Kan extension. The challenge of the Abelian Stabilizer Problem, however, is to find the stabilizer of a specified element. This leads us to a discussion of the Hidden Subgroup Problem in 4.4. We give a slight restatement of this problem in 4.42. Given a set-valued function $f$ defined on a group, there is always some subgroup with the property that $f$ is constant on its subgroups. By Zorn’s Lemma (see 4.415 on page 73) there is, in fact, a maximal one. We use this fact to state The Hidden Subgroup Problem in 4.42 on page 73.

In Chapter 5 we discuss abelian categories. These are mathematical structures in which objects have properties similar to those of abelian groups and in which the collection of morphisms between any two objects is, in fact, an abelian group. We included this investigation in order to explore the mathematical infrastructure required to formulate the Hidden Subgroup Problem and the Fourier transform.

In Chapter 6 we review classical algorithms for computing Kan extensions. We describe their computational complexity and compute several examples. Particular instances of Kan extensions are computed using algorithms that are well-studied in other contexts. Products and coproducts, for example, are simply multiplication and addition of integers. Equalizers and pullbacks are computed by search algorithms. Coequalizers can be viewed as calculation of the connected components of a particular graph.

In Chapter 7 of this report we discuss quantum algorithms for computing Kan extensions and the potential for calculating these quantities more effectively with quantum computers. To make deliberate progress on this challenging problem, we exploited the facts that all right Kan extensions can be constructed from two particular cases (see 2.813 and 2.564) called products and equalizers, and that all left Kan extensions can be constructed from two other particular cases (see 2.824 and 2.661), called coproducts and coequalizers. We further simplify the problem by restricting our Kan extension calculations to the simplicial category $\Delta$ which is equivalent to the category $\text{Set}_f$ of finite sets 3.2. In this context, the calculation of categorical products reduces to multiplication of integers. In 7.111 we discuss quantum circuits for integer multiplication.

Equalizers may be constructed from products and another kind of right Kan extension called a pullback. In 7.131 we relate the calculation of pullbacks to that of claws of a pair of functions. This observation yields the theorem 7.132: Quantum computers can not improve upon classical (probabilistic) complexity of exact pullback calculations. Then using the construction we establish 7.133 Quantum computers can not improve upon classical (probabilistic) complexity of exact equalizer calculations. It follows that quantum computers can not improve upon classical complexity for computing general, right Kan extensions.

In the context of the simplicial category $\Delta$, the calculation of categorical coproducts reduces to addition of integers. In 7.21 we discuss quantum circuits for integer addition. The input data to the coequalizer calculation is equivalent to specifying a particular kind of graph. In 7.221 we describe the coequalizer calculation as computing a dominating set on the transitive closure of this graph. The general dominating set problem is NP-complete. Coequalizers are a particular subclass of problems. In 7.221 we discuss the possibility of solving the Dominating Set Problem via a quantum computer.
In Chapter 8 of this report we discuss the design of our Java implementation of the classical and quantum algorithm implementations. For both classical and quantum algorithms, we developed XML schema for input and output data types including finitely-presented categories and functors. We used the JAXB compiler to automatically generate the data exchange code. See Figure 8.1 on page 103. We implemented quantum circuits using the Apache commons library (see http://commons.apache.org/math) for doing complex matrix calculations. As a consequence of our direct implementation of circuits using complex matrices, the number of wires in our Java circuits must be small. Larger circuits can be represented using the Quipper programming language that we discuss in Chapter 9. As discussed below, the source code and documentation are available in other components of our final deliverables.

In Chapter 9 of this report we use discuss the use of quantum programming languages to implement Kan extension algorithms. In Section 9.1, we describe Haskell implementations of several classical algorithms for computing Kan extensions. Although Haskell is not a quantum programming language, the Quipper language is embedded in Haskell and can (in theory) convert Haskell programs into reversible circuits. Quipper is a quantum programming language that is under development under the IARPA Quantum Computer Science program. In Section 9.2, we discuss use of the abstraction mechanisms and dedicated quantum data types that are available in Quipper. QuaFL is another quantum programming language that is being developed through the Quantum Computer Science program. In Section 9.3, we discuss use of the abstraction mechanisms and dedicated quantum data types that are available in QuaFL.

In Chapter 10 we discuss Kan liftings.

In Chapter 11 we discuss abstract homotopy theory.

Appendix A of this report is an outline of the axioms of set theory. We use this material in Chapter 2.

Appendix B gives sample input XML files for and output from our Java implementation of the Carmody-Walters Algorithm. The example generates all the morphisms of a finite category from a finite presentation of that category. See 6.36 for a discussion of Kan extensions of this type.

Appendix C includes Haskell modules for computing particular left and right Kan extensions. We have included implementations for terminal and initial objects, products and coproducts, equalizers and coequalizers, and pullbacks. The functions and data types defined in these two appendices are suitable for processing by the Quipper engine to produce reversible circuits.

Appendix D describes the development and documentation tools used in this project.

The Bibliography includes references to the books, articles and other resources that we used. Most of those for which we could obtain electronic copies are included in the references/ directory of the final contract deliverables.

**Source Code:** Source code for software produced under the contract is available as a pdf document and as source code files. The document D11PC20232-Java-Source.pdf includes the Java source code. Appendix C includes the Haskell source code. Both Java and Haskell have mechanisms for automatically generating html files from source code documentation. These files should serve as the starting point for programmers seeking to use or modify our code. The Java documentation is available in the directory java/jdoc/index.html that is included in the final contract deliverables.

**Reports and Presentations:** Our monthly technical and financial reports as well as the briefs that we presented are respectively included in the reports/ and briefs/ directories of the final contract deliverables.
Acknowledgments: The authors would like to thank Dr. Mark Heiligman of IARPA for sharing his time and insights with us and for supporting this research. Thanks to Kellie Cloud of IARPA and Stephanie Hunter, Lawrence Carter and Ken Roch of the Department of Interior National Business Center for their time and assistance on contractual matters. We also thank Dr. Peter Selinger of Dalhousie University for providing us with Quipper and answering our questions about the software. Finally, we thank Rich Lazarus and Marcus Silva of BBN for providing us with documentation for QuaFL.

Contact Information: This report and the software described herein are freely-available from the authors. For inquiries, contact Ralph Wojtowicz at ralphw@bakermountain.org.
# Contents

**Executive Summary** ................................. i

**Statement of Work** ................................. xiii

**Guide to Notation** ................................. xv

**List of Code Listings** .............................. xvii

**List of Figures** ..................................... xix

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

2 The Language of Categories .......................... 3
   2.1 Categories ......................................... 3
   2.11 Small Categories .................................. 3
   2.12 Examples ......................................... 4
   2.13 Locally-Small Categories ........................ 4
   2.14 Theory of Categories .............................. 5
   2.15 An Alternative Presentation of the Theory .... 6
   2.16 Two-Sorted Theory of Categories ................. 7
   2.17 Proto-Categories .................................. 8
   2.18 Founded Categories ............................... 9
   2.19 Dual Categories .................................. 9
   2.2 Special Morphisms ................................ 10
   2.21 Monomorphisms .................................. 10
   2.22 Epimorphisms .................................... 10
   2.23 Isomorphisms .................................... 11
   2.3 Functors ........................................... 12
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.31</td>
<td>Overview</td>
<td>12</td>
</tr>
<tr>
<td>2.32</td>
<td>Definition</td>
<td>13</td>
</tr>
<tr>
<td>2.33</td>
<td>Identity Functors</td>
<td>13</td>
</tr>
<tr>
<td>2.34</td>
<td>Forgetful Functors</td>
<td>14</td>
</tr>
<tr>
<td>2.35</td>
<td>Dual Functors</td>
<td>15</td>
</tr>
<tr>
<td>2.36</td>
<td>Composition of Functors</td>
<td>15</td>
</tr>
<tr>
<td>2.37</td>
<td>Faithful Functors</td>
<td>15</td>
</tr>
<tr>
<td>2.38</td>
<td>Reflection of Monomorphisms</td>
<td>16</td>
</tr>
<tr>
<td>2.39</td>
<td>Full Functors</td>
<td>16</td>
</tr>
<tr>
<td>2.3a</td>
<td>Reflection of Isomorphisms</td>
<td>17</td>
</tr>
<tr>
<td>2.3b</td>
<td>Conservative Functors</td>
<td>17</td>
</tr>
<tr>
<td>2.3c</td>
<td>Separators in a Category</td>
<td>17</td>
</tr>
<tr>
<td>2.3d</td>
<td>Generators in a Category</td>
<td>17</td>
</tr>
<tr>
<td>2.4</td>
<td>Natural transformations</td>
<td>18</td>
</tr>
<tr>
<td>2.41</td>
<td>Transformations</td>
<td>18</td>
</tr>
<tr>
<td>2.42</td>
<td>Dual Natural Transformations</td>
<td>18</td>
</tr>
<tr>
<td>2.43</td>
<td>Composition of Natural Transformations</td>
<td>19</td>
</tr>
<tr>
<td>2.44</td>
<td>Products of Natural Transformations</td>
<td>20</td>
</tr>
<tr>
<td>2.45</td>
<td>Natural Isomorphisms</td>
<td>21</td>
</tr>
<tr>
<td>2.46</td>
<td>Equivalences of Categories</td>
<td>22</td>
</tr>
<tr>
<td>2.47</td>
<td>Functor Categories</td>
<td>23</td>
</tr>
<tr>
<td>2.48</td>
<td>Comma Categories</td>
<td>25</td>
</tr>
<tr>
<td>2.49</td>
<td>Yoneda Lemma</td>
<td>26</td>
</tr>
<tr>
<td>2.5</td>
<td>Limits</td>
<td>30</td>
</tr>
<tr>
<td>2.51</td>
<td>Cone Morphisms</td>
<td>30</td>
</tr>
<tr>
<td>2.52</td>
<td>Terminal Objects</td>
<td>30</td>
</tr>
<tr>
<td>2.53</td>
<td>Products</td>
<td>31</td>
</tr>
<tr>
<td>2.54</td>
<td>Equalizers</td>
<td>32</td>
</tr>
<tr>
<td>2.55</td>
<td>Pullbacks</td>
<td>34</td>
</tr>
<tr>
<td>2.56</td>
<td>Complete Categories</td>
<td>36</td>
</tr>
<tr>
<td>2.57</td>
<td>Functors Preserve Cones</td>
<td>37</td>
</tr>
<tr>
<td>2.6</td>
<td>Colimits</td>
<td>38</td>
</tr>
<tr>
<td>2.61</td>
<td>Functors Preserve Cocones</td>
<td>38</td>
</tr>
<tr>
<td>2.62</td>
<td>Initial Objects</td>
<td>39</td>
</tr>
</tbody>
</table>
4.32 Stabilizers .................................................... 70
4.33 Orbits ......................................................... 71
4.34 Kan Extension ................................................. 71
4.4 The Hidden Subgroup Problem ................................ 72
  4.41 Constant Functions on Cosets ................................ 72
  4.42 Statement of Problem ......................................... 73
  4.43 The Hidden Subgroup Problem as a Kan Extension ............ 74
  4.44 Examples .................................................... 76
  4.45 Quantum Algorithms in Group Theory ......................... 78

5 Abelian Categories ............................................. 79
  5.1 Zero Objects and Kernels .................................... 79
    5.11 Zero Objects and Morphisms ............................... 79
    5.12 Kernels .................................................. 81
    5.13 Cokernels ............................................... 81
  5.2 Additive Categories ........................................ 82
    5.21 Preadditive Categories .................................... 82
    5.22 Additive Categories .................................... 83
  5.3 Abelian Categories ......................................... 83
    5.31 Special Morphisms in Abelian Categories .................. 83
    5.32 Hidden Subgroup Problems in Abelian Categories .......... 85

6 Classical Algorithms for Computing Kan Extensions ............. 87
  6.1 The Todd-Coxeter Coset Enumeration Procedure ................ 87
  6.2 The Carmody-Walters Algorithm ............................. 87
  6.3 Examples ................................................ 89
    6.31 Products ............................................... 90
    6.32 Equalizers ............................................. 90
    6.33 Pullbacks ............................................. 91
    6.34 Coproducts ............................................ 91
    6.35 Coequalizers .......................................... 92
    6.36 Morphisms of a Category ................................ 93
    6.37 Orbits of Discrete Iterators ............................. 94
    6.38 The Todd-Coxeter Coset Enumeration Procedure ............ 95
## Contents

7 Quantum Algorithms for Computing Kan Extensions 97

7.1 Right Kan Extensions ........................................ 97
7.11 Products ..................................................... 97
7.12 Equalizers .................................................... 98
7.13 Pullbacks ...................................................... 98
7.14 General Right Kan Extensions .............................. 100

7.2 Left Kan Extensions ........................................... 100
7.21 Coproducts .................................................... 100
7.22 Coequalizers .................................................. 101

8 Java Implementation ........................................ 103

8.1 Software Design .............................................. 103
8.2 Input and Output Files ....................................... 104
8.21 Categories ................................................... 104
8.22 Functors ..................................................... 107
8.23 Actions ........................................................ 109

8.3 Java Packages ................................................ 111
8.4 Examples ........................................................ 112
8.41 Orbits .......................................................... 112
8.42 Coset Enumeration .......................................... 113

9 Quantum Programming Languages ...................... 115

9.1 Haskell Implementation ................................... 115
9.11 Right Kan Extensions .................................... 115
9.12 Left Kan Extensions ...................................... 119
9.13 Todd-Coxeter Procedure ................................ 122

9.2 Quipper Implementation .................................. 122
9.21 Quantum Circuits .......................................... 122
9.22 Language Features ......................................... 123

9.3 QuaFL .......................................................... 124
9.31 Basic Data Types ........................................... 124
9.32 Abstraction Mechanisms ................................. 124

10 Kan Liftings .................................................. 127

10.1 Kan Extensions .............................................. 127
10.2 Kan Liftings ................................................................. 128
  10.21 Kan Liftings as Limits and Colimits ......................... 130
  10.22 Examples of Kan lifting ........................................ 132

11 Abstract Homotopy Theory ............................................. 135
  11.1 Homotopy Kan Extensions and Liftings ....................... 137
  11.2 Cat as a Closed Model Category ................................. 139
    11.21 Historical Background ...................................... 139
    11.22 Related References ......................................... 141
    11.23 Closed Model Categories .................................. 142
    11.24 A Closed Model Structure on Cat ......................... 143
  11.3 Conclusions .......................................................... 149

A Axioms of set theory .................................................... 151
  A.1 Axiom of Extent ................................................... 151
  A.2 Power Set Axiom .................................................. 152
  A.3 Union Axiom ....................................................... 152
  A.4 Axiom of Infinity ................................................ 153
  A.5 Axiom of Replacement ......................................... 153
  A.6 Axiom of Foundation ............................................ 155

B Benchmark Input/Output ............................................... 157

C Haskell Implementations of Particular Kan Extensions ......... 183
  C.1 QKan.Classical.RightKanExtensions .............................. 183
  C.2 QKan.Classical.LeftKanExtensions .............................. 185

D Development and Documentation ...................................... 187
  Bibliography ................................................................ 189
  Index ........................................................................... 197
Statement of Work

**Performer**: Baker Mountain Research Corporation with Brooklyn College as subcontractor

**Project Title**: Quantum Kan Extensions and Applications

**Award Instrument Number**: Contract D11PC20232

**Period of Performance**: 9/26/2011 through 12/31/2012 (extended)

**Research Goals**: The Kan extension formula is a deep mathematical result that computes a diverse array of structures including fixed points, symbolic dynamics, and periodic orbits of dynamic systems; quantifiers and all other operations of first- and higher-order logics; tensor products of vector spaces; Gröbner bases; and composition of links in heterogeneous network models. This culminates in the assertion of Saunders Mac Lane (1972) that “All concepts are Kan extensions.” In our proposal, moreover, we demonstrated that Deutsch’s Algorithm, the Deutsch-Jozsa Algorithm, Simon’s Periodicity Algorithm, the Abelian Stabilizer Algorithm and other examples of the Hidden Subgroup Problem can be formulated as Kan extensions.

Algorithms that compute particular Kan extensions are valuable tools in their fields. The Todd-Coxeter coset enumeration algorithm (1936), for example, was used manually for group-theoretic research prior to its first computer implementation in 1953. It is now implemented in many computer algebra systems. The Knuth-Bendix term-rewriting procedure (1967) is a generalization that has applications to logic and abstract algebra. The Carmody-Leeming-Walters algorithm for computing Kan extensions (1991) subsumes these procedures, unifies and clarifies the ideas behind them and increases their range of applicability.

The computational complexities of the Todd-Coxeter and Knuth-Bendix procedures are not well-understood, hence, neither is complexity of the Kan extension algorithm which generalizes both. Particular examples of coset enumeration are known not to be computable (i.e. the algorithm never terminates in these cases). Even for finite cases, however, it was shown by Sims (1994) that there is no polynomial computational space bound for the Todd-Coxeter algorithm.

The goals of our effort are to:

- Research, implement and analyze the classical algorithm for computing Kan extensions;
- Research, develop, implement and analyze a quantum algorithm for computing Kan extensions;
- Research variations of Kan extensions, such as Kan liftings, and applications of these mathematical concepts to quantum computing; and
- Research, develop and implement known quantum algorithms as Kan extensions and as variants of such. From this perspective, try to find other new quantum algorithms.
The combination of factors:

1. The mathematical significance and ubiquity of Kan extensions,
2. The challenges posed by instances of the classical algorithm, and
3. Connection between Kan extensions and known quantum algorithms

makes the research of quantum Kan extension algorithms a meaningful and potentially high-payoff activity. We anticipate success of our approach because of parallelism inherent in the Kan extension algorithm tables, entanglement between coincidences that occur in these tables, and symmetries in the mathematical definitions.

**Description of the Technical Approach:** Our approach is to 1) formulate and analyze a quantum algorithm for computing Kan extensions, 2) implement the algorithm in a quantum simulation language, 3) research extensions to the homotopy case and to Kan liftings, and 4) describe theoretical and practical applications. We anticipate success of the quantum approach because of parallelism inherent in the Kan extension algorithm tables, entanglement between coincidences that occur in these tables, and symmetries in the mathematical definitions.

The 12 month contract includes the following milestones and deliverables.

1. Monthly progress reports
2. Implementation of the classical algorithm
3. Formulation of quantum Kan extensions algorithm
4. Report on Kan liftings and applications to Grover-like algorithms
5. Report on homotopy Kan extensions and applications
6. Implementation using quantum simulation software
7. Final Report

Dependencies among the tasks are illustrated in the figure below.

```
2 → 3 → 6
   ↓   ↓
  4 → 7
  |   |
 5
```

Task 3 is the only one of the seven that presents a significant technical risk. We define completion of Task 7, subsequent to completion of all other tasks, as the project exit criterion.

Baker Mountain will serve as the prime contractor with CUNY Brooklyn College as a subcontractor. Each team member will contribute to all 7 tasks. Baker Mountain will take primary responsibility for tasks 2 and 6. CUNY will take primary responsibility for tasks 4 and 5. Responsibility will be equally shared in tasks 1, 3, and 7. In addition, Baker Mountain will be responsible for setting up software and document repositories (using SVN).

All software source code and compiled code developed under Tasks 2 and 6 and any required libraries will be delivered to the government. The reports produced under Tasks 1, 4, 5, and 7 will be provided to the government electronically as pdf files.
Guide to Notation

New terms are underlined when defined. Bibliography citations appear in brackets such as [102]. Cross references are indicated with a chapter, section, subsection, or subsubsection number such as 2.811. Sections are numbered sequentially in an alphanumeric format: 87.1, 87.2, ..., 87.9, 87.a, ..., 87.f, for example, indicate the first fifteen sections of chapter 87. Nesting of subsections and subsubsections is indicated with appended characters: 87.11, 87.12, ..., 87.1a, 87.1b, ... are subsections of 87.1 and 87.111, 87.112, ..., 87.11a, ... are subsubsections of 87.11. This concise notation is adapted from [69].

An assertion to be proved is not labeled as a theorem, lemma, or corollary, for example, but is typed in a slanted font:

3.122 Set is complete, hence, cartesian.

and is followed by a justification of the claim:

Because: it has equalizers and small products.

A black box at the right of a page indicates the end of a proof.

Naming of a category after its objects is indicated by the phrase: ‘category of . . . ’ (e.g. the category of sets) while naming after its morphisms is indicated by: ‘category composed of . . . ’ (e.g. the category composed of stochastic matrices).

A diagram such as \[ A \xrightarrow{u} B \xrightarrow{v} C \xrightarrow{w} D \] is described as in a category \( C \) if the nodes are labels for objects of \( C \) and the arrows are labels for morphisms. It is also an abbreviation for certain source-target information. A diagram in a category is said to commute if all composite paths between fixed nodes are equal. All diagrams are assumed to commute unless indicated otherwise.

\[ A \xrightarrow{u} B \]

\[ x \]

\[ y \]

\[ v \]

\[ C \]

\[ w \]

\[ D \]

\[ v \circ u = w \circ x, \quad w \circ y = v, \quad \text{and} \quad y \circ u = x \]

in addition to the source-target information. A puncture mark + removes one equation.

\[ A \xrightarrow{u} B \]

\[ x \]

\[ + \]

\[ v \]

\[ C \]

\[ w \]

\[ D \]
means
\[ v \circ u = w \circ x \quad \text{and} \quad y \circ u = x \]
while \( X \xrightarrow{u} B \) still implies \( v \circ u = w \circ x \). Commutativity of \( A \xrightarrow{f} B \xrightarrow{g} C \) means \( f = id_a \). A puncture does not assert the negation of an equation. \( A \xrightarrow{f} B \xrightarrow{m} C \) means \( m \circ f = m \circ g \) but asserts neither \( f = g \) nor \( f \neq g \).

\( \mathbb{N} \) denotes the set of natural numbers \( \{0, 1, \ldots\} \), \( \mathbb{Z} \) is the set of integers, \( \mathbb{R} \) is the set of reals, and \( \mathbb{R}^+ \) is the set of nonnegative reals. \( n \in \mathbb{N} \) is identified with the set \( \{0, 1, \ldots, n-1\} \).

If \( g : \mathbb{N} \to \mathbb{N} \) is a function, then we define
\[
\Theta(g) = \{ f : \mathbb{N} \to \mathbb{N} \mid \text{there exist } c_1 > 0, c_2 > 0 \text{ and } n_0 \text{ for which } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0 \}
\]
and say that \( g \) is an asymptotically tight bound on the members \( \Theta(g) \). We define
\[
O(g) = \{ f : \mathbb{N} \to \mathbb{N} \mid \text{there exist } c > 0 \text{ and } n_0 \text{ for which } 0 \leq f(n) \leq c g(n) \text{ for all } n \geq n_0 \}
\]
and say that \( g \) is an asymptotic upper bound on the members of \( O(g) \). We define
\[
\Omega(g) = \{ f : \mathbb{N} \to \mathbb{N} \mid \text{there exist } c > 0 \text{ and } n_0 \text{ for which } 0 \leq c g(n) \leq f(n) \text{ for all } n \geq n_0 \}
\]
and say that \( g \) is an asymptotic lower bound on the members of \( \Omega(g) \). We define
\[
o(g) = \{ f : \mathbb{N} \to \mathbb{N} \mid \text{for any } c > 0, \text{ there exists an } n_0 > 0 \text{ for which } 0 \leq f(n) < c g(n) \text{ for all } n \geq n_0 \}
\]
See [50] for examples and theorems involving \( \Theta, \Omega, O \) and \( o \).
# List of Code Listings

1. Pseudo-code for the Todd-Coxeter algorithm ........................................ 69
2. Pseudo-code for the Carmody-Walters algorithm .................................. 88
3. XML schema `category-schema.xsd` for describing finitely-presented categories .......... 105
4. Instance document `categoryTest.xml` for XML schema `category-schema.xsd` ...... 106
5. XML schema `functor-schema.xsd` for describing functors ............................ 108
6. XML schema `action-schema.xsd` for describing category actions ................. 110
7. Instance document for the XML schema `action-schema.xsd` ..................... 111
8. XML document `nat.xml` implementing $\mathbb{N}$ .................................. 112
9. XML document `one.xml` implementing a category with one morphism .......... 112
10. XML document `F.xml` implementing a functor from $\mathbb{N}$ to $\mathcal{B}$ ........... 113
11. Terminal object implemented in Haskell ............................................ 116
12. Products implemented in Haskell ...................................................... 116
13. Equalizers implemented in Haskell ................................................... 117
14. Pullbacks implemented in Haskell from products and equalizers ............... 118
15. Initial object implemented in Haskell ................................................ 119
16. Coproducts implemented in Haskell .................................................. 120
17. Coequalizers implemented in Haskell ................................................ 121
18. Coproduct circuit implemented in Quipper ........................................ 125
List of Figures

1 Quantum algorithms in the context of Kan extensions ........................................... i
6.1 Graphical interpretation of the Carmody-Walters algorithm ................................. 89
6.2 Example of a discrete iterator .............................................................................. 94
6.3 Category presentations for computing orbits of discrete iterators by Kan extension .... 94
6.4 Discrete iterator with numbered states ................................................................. 94
8.1 Java implementation design ................................................................................. 103
8.2 Generators and relations for the finitely-presented category of Listing 4 ............... 106
9.1 Quantum coproduct circuit implemented in Quipper .......................................... 123
Chapter 1

Introduction

Despite the abstract and layered nature of its definitions and theorems, category theory is a highly computational subject. Adjoints, Kan extensions and monads are different expressions of a common general structure into which a diverse family of mathematical constructions organize themselves. This family includes addition, multiplication, upper and lower bounds in ordered sets, cartesian and tensor products, disjoint unions, orbits and fixed points of discrete and continuous-time dynamic systems, syntax and semantics of the logical quantifiers, free groups and vector spaces. We refer to reader to [17, 25, 87, 102] for more examples. In Chapter 2 we present those components of the language of categories that we use throughout this report and we describe the conversion between adjoints and Kan extensions. We refer the reader to [25, 102] for the conversion between monads and the other two.

This project grew out of casual discussions between the two investigators over several years. Both did their doctoral research in category theory. Dr. Yanofsky later became interested in quantum computing co-authored an introductory text [136]. The idea behind this project was that since the family of mathematical objects that arise as Kan extensions (adjoints/monads) is so diverse, it would be useful to explore the potential application of quantum computers to calculating them.

To break the problem into manageable pieces, we exploited the fact that Kan extensions come in two flavors, left and right, and that each class can be constructed from two special cases. We focused on algorithms for computing products and equalizers which are two kinds of right Kan extensions from which all others can be constructed. This led us to research algorithms for pullbacks, another right Kan extension, since quantum algorithms related to this construction were known and since equalizers can then be obtained from products and pullbacks. We then researched algorithms for computing coproducts and coequalizers which are left Kan extensions which give all others.

What we found is that for several general classes of Kan extensions, quantum computers can not offer superior performance on exact calculations. Another class, that of coequalizers, however is a special case of the dominating set problem which is known to be NP-complete. The Hidden Subgroup Problem and quantum search are two classes of quantum algorithms for which quantum computers do offer superior performance. The former computes a Kan extension by exploiting the rich algebraic structure available in the context of abelian groups. This exploration, therefore, reveals three areas in which to search for new quantum algorithms: (1) calculations that involve coequalizers, (2) calculations that can exploit algebraic structure and (3) Kan liftings. Shor’s Algorithm is located in area (2). Perhaps the next general class of quantum algorithms will involve the rich algebraic and order structure available in categories of lattices.
Chapter 2

The Language of Categories

Defining categories is complicated by the need to address foundational issues (see appendix A) and to provide a definition appropriate for the target audience. This chapter provides four definitions and a general technique for presenting examples.

2.1 Categories

Defining categories is complicated by the need to address foundational issues (see appendix A) and to provide a definition appropriate for the target audience. This chapter provides four definitions and a general technique for presenting examples.

2.11 Small Categories. A small category $\mathcal{C}$ consists of

i) a set $|\mathcal{C}|$ of objects;

ii) for each pair $(A, B)$ of objects, a set $\mathcal{C}(A, B)$ of morphisms;

iii) for each object $A$, an identity morphism $id_A \in \mathcal{C}(A, A)$;

iv) a composition function $\circ : \mathcal{C}(B, C) \times \mathcal{C}(A, B) \to \mathcal{C}(A, C)$ for each triple $(A, B, C)$ of objects:

\[
\begin{array}{ccc}
A & \xrightarrow{f} & B \\
\downarrow{g} & \searrow{g} & \\
C. & & \\
\end{array}
\]

These data are subject to the following axioms:

i) if $f \in \mathcal{C}(A, B)$ then $f \circ id_A = f$ and $id_B \circ f = f$:

\[
\begin{array}{ccc}
A & \xrightarrow{id_A} & A \\
\downarrow{f} & \searrow{f} & \\
B & & \\
\end{array}
\quad
\begin{array}{ccc}
A & \xrightarrow{f} & B \\
\downarrow{id_B} & \searrow{id_B} & \\
B & & \\
\end{array}
\]
ii) if \( f \in \mathcal{C}(A, B) \), \( g \in \mathcal{C}(B, C) \) and \( h \in \mathcal{C}(C, D) \) then \( h \circ (g \circ f) = (h \circ g) \circ f \).

\(|\mathcal{C}|\) and \(|\mathcal{C}|\) respectively denote the set of objects and the set of morphisms of the small category \( \mathcal{C} \). 

\( A \xrightarrow{f} B \) indicates that \( f \in \mathcal{C}(A, B) \) is a morphism of \( \mathcal{C} \). \( \Box f = A \) is the source or domain of \( f \) while \( f \Box = B \) is the target or codomain of \( f \).

### 2.12 Examples

The set \( \mathbb{R} \) of real numbers may be viewed as a category \( \mathcal{C} \) in various ways. For example, let \(|\mathcal{C}| = \{\ast\}\) be a one-point set, let \( \mathcal{C}(\ast, \ast) = \mathbb{R} \), let \( \text{id}_{\ast} = 0 \), and let \( \circ = + \). That is, the additive structure on \( \mathbb{R} \) induces a category having \( \mathbb{R} \) as its set of morphisms and addition as composition.

Let \(|\mathcal{C}| = \mathbb{R} \), let \( \mathcal{C}(x, y) = \begin{cases} \{\ast\} & \text{if } x \leq y; \\ \phi & \text{otherwise.} \end{cases} \) That is, there is a unique morphism \( x \xrightarrow{\ast} y \) iff \( x \leq y \).

Identities and composites are uniquely determined by this definition since \( x \leq x \) for all \( x \in \mathbb{R} \) and \( x \leq y \leq z \) implies \( x \leq z \). Other finite examples can be found in [128].

### 2.13 Locally-Small Categories

Definition 2.11 requires the collection of objects of a small category to be a set. This precludes sets and functions from forming a small category, however, since the collection of all sets is not itself a set (see [121]). The collection of all sets in some model of the axioms of set theory is a class. Every set is a class but there are classes which are not sets. Axioms of set theory are described in Appendix A. [102] describes set-theoretic foundations of category theory.

A locally small category consists of

i) a class \(|\mathcal{C}|\) of objects;

ii) for each pair \((A, B)\) of objects, a set \( \mathcal{C}(A, B) \) of morphisms;

iii) for each object \( A \), an identity morphism \( \text{id}_A \in \mathcal{C}(A, A) \);

iv) a composition function \( \circ : \mathcal{C}(B, C) \times \mathcal{C}(A, B) \to \mathcal{C}(A, C) \) for each triple \((A, B, C)\) of objects:

\[
\begin{array}{ccc}
A & \xrightarrow{f} & B \\
\downarrow{g \circ f} & & \downarrow{g} \\
C, & & \\
\end{array}
\]

These data are subject to axioms i) – ii) of 2.11.

The category \( \text{Set} \), for example, has the class of sets as its objects and functions as its morphisms. It is locally-small but not small. Composition and identity morphisms are function composition and identity functions. Vector spaces (over any fixed field \( k \)) and linear transformations organize themselves into a category \( \text{Vect}_k \). Given a directed, multi-graph \( G \), the set of paths in \( G \) can be construed as a category if we associate an identity (or empty) loop with each vertex. Partial functions form the morphisms of a category having the same objects as \( \text{Set} \). Any partially-ordered set (or poset) is a category. Its objects are the elements of \( X \). There is a unique morphism \( x \to x' \) iff \( x \leq x' \). Composition is given by transitivity of \( \leq \) and identity morphisms are given by reflexivity.
2.131 **Finitely Presented Categories.** A category is finitely presented if it has a finite set of objects and its all morphisms can be generated as composites of a fixed, finite set of morphisms by imposing elements of a finite set of relations. In other words, a finitely-presented category consists of a directed multi-graph together with a finite set of relations between paths. It follows that such a category is small. To be specific, a path is a finite sequence \( \rho = f_0, f_1, \ldots, f_n \) of morphisms for which the codomain of \( f_{i-1} \) is the domain of \( f_i \) for \( i \in \{1, \ldots, n\} \). The **domain** of \( \rho \) is the domain of \( f_0 \) and the **codomain** of \( \rho \) is the codomain of \( f_n \). A **relation** is a pair \((\rho, \gamma)\) of paths which have the same domain and the same codomain.

2.14 **Theory of Categories.** The theory of categories may be presented using the following axioms involving one sort, Morphisms; binary relation symbols, 

Source \( \leftrightarrow \) Morphisms, Morphisms and Target \( \leftrightarrow \) Morphisms, Morphisms

in addition to equality; and one ternary relation symbol,

Composite \( \leftrightarrow \) Morphisms, Morphisms, Morphisms.

The grouping of axioms below is from [100]. The syntax is adjusted to fit that described in [87]. The Rules for Implication have been used to convert the first-order formulae \( E_2, D_2, D^*_2, \) and \( C_2 \) from [100] to the regular formulae \( E_3 – E_4, S_3 – S_4, T_3 – T_4, \) and \( C_3 – C_4 \) shown below.

**Axioms of Existence**

E1. \((\top \vdash_x (\exists a) \text{Source}(x, a))\)

E2. \((\top \vdash_x (\exists b) \text{Target}(x, b))\)

E3. \((\exists u) \text{Composite}(x, y, u) \vdash_{x,y} (\exists b) \text{Target}(x, b) \land \text{Source}(y, b))\)

E4. \((\exists b) \text{Target}(x, b) \land \text{Source}(y, b) \vdash_{x,y} (\exists u) \text{Composite}(x, y, u))\)

**Axioms of Uniqueness**

U1. \((\text{Source}(x, a) \land \text{Source}(x, a') \vdash_{x,a,a'} (a = a'))\)

U2. \((\text{Target}(x, b) \land \text{Target}(x, b') \vdash_{x,b,b'} (b = b'))\)

U3. \((\text{Composite}(x, y, u) \land \text{Composite}(x, y, u') \vdash_{x,y,u,u'} (u = u'))\)

**Axioms of Domain**

S1. \((\text{Source}(x, a) \vdash_{x,a} \text{Source}(a, a))\)

S2. \((\text{Target}(x, b) \vdash_{x,b} \text{Source}(b, b))\)

S3. \((\text{Composite}(x, y, u) \land \text{Source}(u, a) \vdash_{x,y,u,a} \text{Source}(x, a))\)

S4. \((\text{Composite}(x, y, u) \land \text{Source}(x, a) \vdash_{x,y,u,a} \text{Source}(u, a))\)

**Axioms of Codomain**

T1. \((\text{Source}(x, a) \vdash_{x,a} \text{Target}(a, a))\)

T2. \((\text{Target}(x, b) \vdash_{x,b} \text{Target}(b, b))\)

T3. \((\text{Composite}(x, y, u) \land \text{Target}(u, b) \vdash_{x,y,u,b} \text{Target}(y, b))\)
T4. \((\text{Composite}(x, y, u) \land \text{Target}(y, b) \vdash_{x,y,u,b} \text{Target}(u, b))\)

Axioms of Composition

C1. \((\text{Source}(x, a) \vdash_{x,a} \text{Composite}(a, x, x))\)
C2. \((\text{Target}(x, b) \vdash_{x,b} \text{Composite}(x, b, x))\)
C3. \((\text{Composite}(x, y, u) \land \text{Composite}(y, z, v) \land \text{Composite}(x, v, t) \vdash_{x,y,u,v,t} \text{Composite}(u, z, t))\)
C4. \((\text{Composite}(x, y, u) \land \text{Composite}(y, z, v) \land \text{Composite}(u, z, t) \vdash_{x,y,u,v,t} \text{Composite}(x, v, t))\)

Pronunciations of atomic formulae are

- the source of \(x\) is \(a\) for \(\text{Source}(x, a)\);
- the target of \(x\) is \(b\) for \(\text{Target}(x, b)\);
- the composite of \(x\) followed by \(y\) is \(z\) for \(\text{Composite}(x, y, z)\).

A model of this theory is a category.

2.15 An Alternative Presentation of the Theory. Alternatively, the single-sorted theory of categories may be presented using the following axioms involving the function symbols

\[\square \cdot : \text{Morphisms} \rightarrow \text{Morphisms}\]
and ternary relation symbol

\[\text{Composite} \rightarrow \text{Morphisms}, \text{Morphisms}, \text{Morphisms}\]

in addition to the binary relation of equality. The following axioms are from [69] and are adjusted to fit the syntax described in [87].

Axioms of Existence

E1. \((\exists u) \text{Composite}(x, y, u) \vdash_{x,y} (x\square = \square y)\)
E2. \((x\square = \square y) \vdash_{x,y} (\exists u) \text{Composite}(x, y, u)\)

Axiom of Uniqueness

U1. \((\text{Composite}(x, y, u) \land \text{Composite}(x, y, u') \vdash_{x,y,u,u'} (u = u'))\)

Axioms of Domain

S1. \((\top \vdash_{x} (x\square = \square (x\square)))\)
S2. \((\text{Composite}(x, y, u) \land \text{Composite}(x, \square y, v) \vdash_{x,y,u,v} (\square u = \square v))\)

Axioms of Codomain

T1. \((\top \vdash_{x} (\square x = (\square x)\square)))\)
T2. \((\text{Composite}(x, y, u) \land \text{Composite}(x\square, y, v) \vdash_{x,y,u,v} (u\square = v\square))\)

Axioms of Identity
2.1. CATEGORIES

I1. \((\top \vdash \text{Composite}(\Box x, x, x))\)

I2. \((\top \vdash \text{Composite}(x, x, \Box x))\)

Axioms of Composition

C1. \((\text{Composite}(x, y, u) \land \text{Composite}(y, z, v) \land \text{Composite}(x, v, t) \vdash_{x,y,u,v,t} \text{Composite}(u, z, t))\)

C2. \((\text{Composite}(x, y, u) \land \text{Composite}(y, z, v) \land \text{Composite}(u, z, t) \vdash_{x,y,u,v,t} \text{Composite}(x, v, t))\)

Pronunciations of terms and atomic formulae are

- the source of \(x\) for \(\Box x\);
- the target of \(x\) for \(x \Box\);
- the composite of \(x\) followed by \(y\) is \(z\) for \(\text{Composite}(x, y, z)\).

A model of this theory is a category.

2.151 Both sequents

\((\top \vdash_x (\Box (\Box x) = \Box x))\) \text{ and } \((\top \vdash_x ((x \Box) \Box = x \Box))\)

are provable.

Because: the proofs on page 3 of [69] may be adjusted to fit the syntax presented in this report.

2.16 Two-Sorted Theory of Categories. The theory of categories may be presented using two sorts Objects and Morphisms; function symbols

\(id : \text{Objects} \rightarrow \text{Morphisms}, \quad \Box \cdot : \text{Morphisms} \rightarrow \text{Objects}, \quad \text{and} \quad \cdot \Box : \text{Morphisms} \rightarrow \text{Objects};\)

and a ternary relation symbol

\(\text{Composite} \rightarrow \text{Morphisms}, \text{Morphisms}, \text{Morphisms}\)

in addition to the binary relation of equality. The axioms below are adapted from Example D1.1.7 of [87].

Axioms of Existence

E1. \((\exists u) \text{Composite}(x, y, u) \vdash_{x,y,u} (x \Box = \Box y))\)

E2. \((x \Box = \Box y) \vdash_{x,y} (\exists u) \text{Composite}(x, y, u))\)

Axiom of Uniqueness

U1. \((\text{Composite}(x, y, u) \land \text{Composite}(x, y, u') \vdash_{x,y,u,u'} (u = u'))\)

Axioms of Domain

S1. \((\top \vdash_x (\Box id(A) = A))\)

S2. \((\text{Composite}(x, y, z) \vdash_{x,y,z} (\Box x = \Box z))\)
Axioms of Codomain

T1. \( \top \vdash_x (id(A) \square = A) \)
T2. \( \text{Composite}(x, y, z) \vdash_{x,y,z} (y \square = z \square) \)

Axioms of Identity

I1. \( \vdash_x \text{Composite}(id(\square x), x, x) \)
I2. \( \vdash_x \text{Composite}(x, id(\square x), x) \)

Axioms of Composition

C1. \( \text{Composite}(x, y, u) \land \text{Composite}(y, z, v) \land \text{Composite}(x, v, t) \vdash_{x,y,u,v,t} \text{Composite}(u, z, t) \)
C2. \( \text{Composite}(x, y, u) \land \text{Composite}(y, z, v) \land \text{Composite}(u, z, t) \vdash_{x,y,u,v,t} \text{Composite}(x, v, t) \)

Pronunciations of terms and atomic formulae are

- the source of \( x \) for \( \square x \);
- the target of \( x \) for \( x \square \);
- the identity morphism of \( A \) for \( id(A) \);
- the composite of \( x \) followed by \( y \) is \( z \) for \( \text{Composite}(x, y, z) \).

A model of this theory is a category.

2.17 Proto-Categories. It is frequently convenient to describe a category using the notion of protocategory introduced in [69] and also used in [87]. The theory of proto-categories may be presented using two sorts Objects and Proto-Morphisms; a function symbol

\[ id : \text{Objects} \rightarrow \text{Proto-Morphisms} \]

and ternary relation symbols

\[ \text{Arrow} \rightarrow \text{Objects}, \text{Proto-Morphisms}, \text{Objects} \]

and

\[ \text{Composite} \rightarrow \text{Proto-Morphisms}, \text{Proto-Morphisms}, \text{Proto-Morphisms} \]

in addition to equality. The axioms below are adapted from [69].

Axiom of Existence

E1. \( \text{Arrow}(A, x, B) \land \text{Arrow}(B, y, C) \vdash_{x,y,A,B,C} (\exists z) \text{Composite}(x, y, z) \land \text{Arrow}(A, z, C) \)

Axioms of Identity

I1. \( \vdash_A \text{Arrow}(A, id(A), A) \)
I2. \( \text{Arrow}(A, x, B) \vdash_{x,A,B} \text{Composite}(id(A), x, x) \)
I3. \( \text{Arrow}(A, x, B) \vdash_{x,A,B} \text{Composite}(x, id(B), x) \)

Axiom of Associativity
A1. \((\text{Arrow}(A, x, B) \land \text{Arrow}(B, y, C) \land \text{Arrow}(C, z, D) \land \text{Arrow}(A, s, C)) \land (\text{Arrow}(A, t, D) \land \text{Arrow}(B, u, D) \land \text{Arrow}(A, v, D) \land \text{Composite}(x, u, v) \land \text{Composite}(s, z, t) \land \text{Composite}(x, u, v) \land \text{Composite}(y, z, u))\)
\[
\vdash s, t, u, v, x, y, z, A, B, C, D \quad (t = v)
\]

Pronunciations of terms and atomic formulae are
- the identity morphism of \(A\) for \(\text{id}(A)\);
- \(x\) may be construed as going from \(A\) to \(B\) for \(\text{Arrow}(A, x, B)\);
- a composite of \(x\) followed by \(y\) is \(z\) for \(\text{Composite}(x, y, z)\).

A model of these axioms is a proto-category. The arrow relation symbol is also referred to as the source-target predicate. [69] and [87] describe the sense in which a proto-category generates a category.

2.18 Founded Categories. A category \(\mathcal{A}\) is founded on a category \(\mathcal{B}\) if it inherits its morphism and composition rule from \(\mathcal{B}\). See 1.243 on page 9 of [69]. In [87] (pages 6-7 of Volume I), \(\mathcal{A}\) is said to be structured over \(\mathcal{B}\) if this same condition holds.

2.181 Let \(\mathcal{C}\) and \(\mathcal{D}\) be set-valued models of the theory of categories (in particular, \(\mathcal{C}\) and \(\mathcal{D}\) may be small or locally small). \(\mathcal{C}\) is a subcategory of \(\mathcal{D}\) if each object of \(\mathcal{C}\) is an object of \(\mathcal{D}\); \(\mathcal{C}(A, B) \subseteq \mathcal{D}(A, B)\) for each pair of \(\mathcal{C}\)-objects; the domain, codomain, composition, and identity operations in \(\mathcal{C}\) are restrictions of those operations in \(\mathcal{D}\).

2.182 A subcategory \(\mathcal{C}\) of a category \(\mathcal{D}\) is full if \(\mathcal{C}(A, B) = \mathcal{D}(A, B)\) for each pair of \(\mathcal{C}\) objects.

2.19 Dual Categories. The dual category \(\mathcal{C}^\circ\) of a category \(\mathcal{C}\) has: an object \(C^\circ\) for each \(C \in |\mathcal{C}|\), a morphism
\[
\begin{array}{ccc}
C^\circ & \xrightarrow{f^\circ} & D^\circ \\
\downarrow{f^\circ \circ g^\circ} & & \downarrow{g^\circ} \\
E^\circ & \xleftarrow{g^\circ} & D^\circ
\end{array}
\]
for each \(C \xrightarrow{f} D\) in \(\mathcal{C}\), composition \(\circ^\circ\) for which \(f^\circ \circ g^\circ = (g \circ f)^\circ\), that is:
\[
\begin{array}{ccc}
C^\circ & \xrightarrow{f^\circ} & D^\circ \\
\downarrow{f^\circ} & & \downarrow{g^\circ} \\
E^\circ & \xleftarrow{g^\circ} & D^\circ
\end{array}
\]
and an identity morphism \(\text{id}_{C^\circ} = (\text{id}_C)^\circ\) for each object. By definition, \(C^{\circ\circ} = C\) and \(f^{\circ\circ} = f\), and \(\circ^{\circ\circ} = \circ\).

2.191 The dual category \(\mathcal{C}^\circ\) of any category \(\mathcal{C}\) is a category.

Because: \(k^\circ \in \mathcal{C}^\circ(C^\circ, D^\circ)\) implies
\[
\text{id}_{D^\circ} \circ^\circ k^\circ = (\text{id}_{D})^\circ \circ^\circ k^\circ = (k \circ \text{id}_{D})^\circ = k^\circ
\]
and
\[
k^\circ \circ^\circ \text{id}_{C^\circ} = k^\circ \circ^\circ (\text{id}_{C})^\circ = (\text{id}_{C} \circ k)^\circ = k^\circ.
\]
f^\circ \in C^\circ (C^\circ , D^\circ ), g^\circ \in C^\circ (D^\circ , E^\circ ), and h^\circ \in C^\circ (E^\circ , F^\circ ) implies
\[ h^\circ \circ (g^\circ \circ f^\circ ) = (g \circ h )^\circ = (g \circ h )^\circ f^\circ = (h^\circ \circ g^\circ )^\circ f^\circ . \]

2.2 Special Morphisms

As discussed in Section 2.14, the theory of categories may be presented using a single sort: morphisms. Consequently, one aspect of category theory is reformulating the membership-based concepts from set theory using morphisms. In this section we discuss special classes of morphisms that generalize the notions of one-to-one, onto and bijective functions.

2.21 Monomorphisms. Let \( C \) be a category. A nonempty class of morphisms \( \{ T \xrightarrow{f_\alpha} A_\alpha \mid \alpha \} \) is a monomorphism (or is monic) if, given morphisms \( C \xrightarrow{k} T \) and \( C \xrightarrow{k'} T \), \( f_\alpha \circ k = f_\alpha \circ k' \) for each \( \alpha \) implies \( k = k' \). In diagrams,

\[
\begin{array}{ccc}
C & \xrightarrow{k} & T \\
\downarrow{k'} & \nearrow{f_\alpha} & \downarrow{f_\alpha'} \\
A_\alpha' & \Downarrow{f_\alpha} & A_\alpha \\
\end{array}
\quad \text{implies} \quad
\begin{array}{ccc}
C & \xrightarrow{k} & T \\
\downarrow{k'} & \nearrow{f_\alpha} & \downarrow{f_\alpha'} \\
A_\alpha' & \Downarrow{f_\alpha} & A_\alpha \\
\end{array}
\]

A single morphism \( X \xrightarrow{f} Y \) is monic when

\[
\begin{array}{ccc}
X' \xleftarrow{+} & X & \xrightarrow{f} Y \\
\Downarrow{f_\alpha'} & \nearrow{f_\alpha} & \Downarrow{f_\alpha} \\
A_\alpha' & \Downarrow{f_\alpha} & A_\alpha \\
\end{array}
\quad \text{implies} \quad
\begin{array}{ccc}
X' \xleftarrow{+} & X & \xrightarrow{f} Y \\
\Downarrow{f_\alpha'} & \nearrow{f_\alpha} & \Downarrow{f_\alpha} \\
A_\alpha' & \Downarrow{f_\alpha} & A_\alpha \\
\end{array}
\]

\( X \xrightarrow{f} Y \) indicates that \( f \) is monic. Monomorphisms generalize injective functions between sets (see 3.111). Fixed points and other invariant manifolds of dynamic systems are also examples of monomorphisms. Limit cones are a common source of monomorphisms (see 2.513).

2.22 Epimorphisms. A nonempty class of morphisms \( \{ T \xrightarrow{f_\alpha} A_\alpha \mid \alpha \} \) is an epimorphism (or is epic) if, given morphisms \( C \xrightarrow{k} T \) and \( C \xrightarrow{k'} T \), \( k \circ f_\alpha = k' \circ f_\alpha \) for each \( \alpha \) implies \( k = k' \). In diagrams,

\[
\begin{array}{ccc}
A_\alpha & \xrightarrow{f_\alpha} & T \xleftarrow{+} C \\
\downarrow{f_\alpha'} & \nearrow{k} & \downarrow{k'} \\
A_\alpha' & \Downarrow{f_\alpha'} & A_\alpha \\
\end{array}
\quad \text{implies} \quad
\begin{array}{ccc}
A_\alpha & \xrightarrow{f_\alpha} & T \xleftarrow{+} C \\
\downarrow{f_\alpha'} & \nearrow{k} & \downarrow{k'} \\
A_\alpha' & \Downarrow{f_\alpha'} & A_\alpha \\
\end{array}
\]

\( X \xleftarrow{f} Y \) indicates that \( f \) is epimorphic. Epimorphisms generalize surjective functions between sets (see 3.112).
A single morphism $X \xrightarrow{f} Y$ is epic when

$$X \xrightarrow{f} Y \xLeftarrow{g} Y'$$ implies $$X \xrightarrow{f} Y \xLeftarrow{g} Y'.

2.221 A set \(\{f_\alpha\}\) of morphisms in a category \(C\) is monic iff \(\{f_\alpha^\circ\}\) is epic in \(C^\circ\).

\[Because:\ \Rightarrow\text{ if } \{f_\alpha\}\text{ is monic in } C \text{ and } k_1^\circ \circ f_\alpha^\circ = k_2^\circ \circ f_\alpha^\circ \text{ for all } \alpha, \text{ then the definition of } C^\circ \text{ justifies equalities two and four of}\]

$$k_1^\circ \circ f_\alpha^\circ = k_2^\circ \circ f_\alpha^\circ$$
$$f_\alpha \circ k_1 = f_\alpha \circ k_2$$
$$k_1 = k_2$$
$$k_1^\circ = k_2^\circ$$

while the definition of monic justifies equality three.

\[\Leftarrow:\ \text{Similar.}\]

2.23 Isomorphisms. $A \xrightarrow{f} B$ is an isomorphism (or is iso) in a category \(C\) iff there exists $B \xrightarrow{g} A$ for which

\[\begin{array}{ccc}
A & \xrightarrow{f} & B \\
\downarrow{id_A} & & \downarrow{g} \\
A & & B
\end{array}\]

\[\begin{array}{ccc}
B & \xleftarrow{g} & A \\
\uparrow{id_B} & & \uparrow{f} \\
B & & A
\end{array}\]

2.231 $A \xrightarrow{f} B$ is an isomorphism iff there is a unique $B \xleftarrow{f^{-1}} A$ for which $f \circ f^{-1} = id_B$ and $f^{-1} \circ f = id_A$.

\[Because:\ f \circ g = id_B \text{ and } g \circ f = id_A \text{ implies } g = id_A \circ g = f^{-1} \circ f \circ g = f^{-1} \circ \circ id_B = f^{-1}.\]

2.232 $A \xrightarrow{f} B$ is an isomorphism in \(C\) iff $B^\circ \xrightarrow{f^\circ} A^\circ$ is an isomorphism in \(C^\circ\).

\[Because:\ f \circ g = id_B \text{ and } g \circ f = id_A \text{ iff } g^\circ \circ f^\circ = (id_B)^\circ \text{ and } f^\circ \circ g^\circ = (id_A)^\circ \text{ iff } g^\circ \circ f^\circ = id_{B^\circ} \text{ and } f^\circ \circ g^\circ = id_{A^\circ}.\]

2.233 A composite of isomorphisms is an isomorphism.

\[Because:\ A \xrightarrow{\varphi} B \text{ and } B \xrightarrow{\psi} C \text{ isos implies}\]

$$(\varphi^{-1} \circ \psi^{-1}) \circ (\psi \circ \varphi) = \varphi^{-1} \circ \psi^{-1} \circ \psi \circ \varphi = \varphi^{-1} \circ id_B \circ \varphi = \varphi^{-1} \circ \varphi = id_A.$$ Similarily, $(\psi \circ \varphi) \circ (\varphi^{-1} \circ \psi^{-1}) = id_C.$

2.234 Every isomorphism is monic and epic.
Because: if $X \xrightarrow{\varphi} Y$ is iso and $\varphi \circ u = \varphi \circ v$, then
\[ u = id_X \circ u = \varphi^{-1} \circ \varphi \circ u = \varphi^{-1} \circ \varphi \circ v = id_X \circ v = v, \]
hence, $\varphi$ is monic. If $w \circ \varphi = z \circ \varphi$, then
\[ w = w \circ id_Y = w \circ \varphi \circ \varphi^{-1} = z \circ \varphi \circ \varphi^{-1} = z \circ id_Y = z, \]
hence, $\varphi$ is epic.

\[ \blacktriangleleft \]

2.235 A category $C$ is balanced if every morphism which is both monic and epic is an isomorphism.

The converse of 2.234 is false. The category of sets and functions is balanced (3.113), however, the ordered set of reals is not (every morphism is both monic and epic while only identity morphisms are iso).

2.3 Functors

Defining categories was complicated by the needs to address foundational issues and to provide a definition appropriate for the target audience. As a consequence, four definitions of category were provided in ??.

A definition of functor may be formulated for each corresponding type of category.

2.31 Overview. A functor $F : A \rightarrow B$ from a category $A$ to a category $B$ assigns a $B$-object $F(A)$ to each $A$-object $A$ and a $B$-morphism $F(a)$ to each $A$-morphism $a$. These ingredients are subject to the following axioms. (1) $F(id_A) = id_{F(A)}$ and (2) $F(g \circ f) = F(g) \circ F(f)$. The underlying set of a vector space gives the object part of a functor $U : \text{Vect}_k \rightarrow \text{Set}$. The free vector space construction (for a fixed field $k$) gives a functor in the other direction.

Functors $A \xrightarrow{G} B$ are adjoint (with $F$ a left adjoint of $G$ and $G$ a right adjoint of $F$) if: (1) for each pair of objects $(B, A)$ with $B \in |B|$ and $A \in |A|$, there is a bijection $\varphi_{B,A} : A(F(B), A) \rightarrow B(B, G(A))$ and (2) for any $a \in A(A, A')$, $b \in B(B', B)$, and $h \in A(F(B), A)$,
\[ \varphi_{B',A'}(a \circ h \circ F(b)) = G(a) \circ \varphi_{B,A}(h) \circ b. \]

The second condition is equivalent to the assertion that the following diagram is commutative
\[ \begin{array}{ccc}
A(F(B), A) & \xrightarrow{\varphi_{B,A}} & B(B, G(A)) \\
A(F(b), a) \downarrow & & \downarrow B(b, G(a)) \\
A(F(B'), A') & \xrightarrow{\varphi_{B',A'}} & B(B', G(A'))
\end{array} \]

where $A(F(b), a)(h) = a \circ h \circ F(b)$ for any $h \in A(F(B), A)$ and $B(b, G(a))(k) = G(a) \circ k \circ b$ for any $k \in B(B, G(A))$. The free vector space and underlying set functors form an example of an adjunction $\text{Vect} \xrightarrow{U} \text{Set}$. If $F(X)$ is a vector space with basis $X$ and $V$ is a vector space, existence of a bijection
\[ \varphi_{X,V} : \text{Vect}_k(F(X), V) \rightarrow \text{Set}(X, U(V)) \]
follows from the definition of basis. The curry operation on functions is an adjunction

\[
\text{Set} \xrightarrow{\times} \text{Set}
\]

where \(\times\) : \(\text{Set} \rightarrow \text{Set}\) maps each set \(B\) to the cartesian product \(A \times B\) and \([A,\ ] : \text{Set} \rightarrow \text{Set}\) maps each set \(B\) to the set \([A, B]\) of functions \(A \rightarrow B\). The adjunction is given by the following bijection that holds for any sets \(A, X\) and \(Y\).

\[
\varphi_{X,Y} : \text{Set}([A, X], Y) \rightarrow \text{Set}(X, A \times Y)
\]

Functors between pre-ordered sets, viewed as categories, are the same thing as order-preserving functions. In this context \((X, \preceq) \xrightarrow{G} (Y, \preceq)\) are adjoint (with \(F\) left adjoint to \(G\)) iff both inferences (top-to-bottom and bottom-to-top) in the following rule hold.

\[
\begin{align*}
F(x) \preceq y & \quad \text{iff} \quad x \preceq G(y) \\
\end{align*}
\]

2.32 Definition. Let \(\mathcal{B}\) and \(\mathcal{C}\) be small categories. A functor \(F : \mathcal{B} \rightarrow \mathcal{C}\) is a function\(^1\) mapping objects of \(\mathcal{B}\) to objects of \(\mathcal{C}\) and morphisms of \(\mathcal{B}\) to morphisms of \(\mathcal{C}\). This function is subject to the following axioms:

i) \(f \in \mathcal{B}(B, B')\) implies \(F(f) \in \mathcal{C}(F(B), F(B'))\);

\[
\begin{array}{c}
B \\
\downarrow f \\
B'
\end{array}
\rightarrow
\begin{array}{c}
F(B) \\
\downarrow F(f) \\
F(B')
\end{array}
\]

ii) \(B \in |\mathcal{B}|\) implies \(F(1_B) = 1_{F(B)}\);

\[
\begin{array}{c}
B \\
\circlearrowleft \quad id_B \\
\end{array}
\rightarrow
\begin{array}{c}
F(B) \\
\circlearrowleft \quad id_{F(B)} \\
\end{array}
\]

iii) \(f \in \mathcal{B}(B, B')\) and \(f' \in \mathcal{B}(B', B'')\) implies \(F(f' \circ f) = F(f') \circ F(f)\).

\[
\begin{array}{c}
B \\
\downarrow f \\
B'
\end{array}
\rightarrow
\begin{array}{c}
F(B) \\
\downarrow F(f) \\
F(B')
\end{array}
\]

\[
\begin{array}{c}
B \\
\downarrow f' \circ f \\
B''
\end{array}
\rightarrow
\begin{array}{c}
F(B) \\
\downarrow F(f' \circ f) \\
F(B'')
\end{array}
\]

2.33 Identity Functors. For any category \(\mathcal{C}\) there is an identity functor \(id_\mathcal{C} : \mathcal{C} \rightarrow \mathcal{C}\) defined by \(id_\mathcal{C}(C) = C\) and \(id_\mathcal{C}(k) = k\).

Because: \(id_\mathcal{C}\) is well-defined on objects and morphisms. It satisfies the functor axioms.

\(^1\)The meaning of ‘function’ depends on the meaning of ‘model.’
2.34  **Forgetful Functors.** Let $\mathcal{A}$ and $\mathcal{B}$ be categories with $\mathcal{A}$ founded on $\mathcal{B}$. There is a functor $U : \mathcal{A} \to \mathcal{B}$.

*Because: see [69] or [87].*

$U$ is called the forgetful or underlying object functor of $\mathcal{A}$.

2.341  A particular case of 2.34 is the inclusion $\mathcal{B}' \to \mathcal{B}$ of a subcategory into its ambient category.

2.342  Let $\mathcal{C}$ and $\mathcal{D}$ be categories. For $D \in |\mathcal{D}|$ there is a constant functor $\kappa(D) : \mathcal{C} \to \mathcal{D}$ defined by $\kappa(D)(C) = C$ and $\kappa(D)(k) = \text{id}_D$.

2.343  If $\mathcal{A}$ is a locally small category and $A \in |\mathcal{A}|$, there is a functor

\[ \hat{\mathcal{A}} : \mathcal{A}^\circ \to \text{Set} \]

with $\hat{\mathcal{A}}(B^\circ) = \mathcal{A}(B, A)$ on objects. For $b^\circ \in \mathcal{A}^\circ(B^\circ, B_1^\circ)$ and $f \in \mathcal{A}(B, A)$, $\hat{\mathcal{A}}(b^\circ)(f) = f \circ b$.

\[
\begin{array}{c}
B_1 \xrightarrow{b} B \xrightarrow{f} A \\
\end{array}
\]

*Because: $\mathcal{A}$ locally small implies $\hat{\mathcal{A}}$ is well-defined on objects. The diagram illustrates that $\hat{\mathcal{A}}$ is well-defined on morphisms. $\hat{\mathcal{A}}$ preserves identities:*

\[ \hat{\mathcal{A}}(\text{id}_{B^\circ})(f) = f \circ \text{id}_{B} = f. \]

$\hat{\mathcal{A}}$ preserves composites as well:

\[ \hat{\mathcal{A}}(b^\circ \circ b_1^\circ)(f) = f \circ (b_1 \circ b) = (f \circ b_1) \circ b = \hat{\mathcal{A}}(b^\circ)(f \circ b_1) = \hat{\mathcal{A}}(b^\circ) \circ \hat{\mathcal{A}}(b_1^\circ)(f). \]

2.344  If $\mathcal{A}$ is a locally small category and $A \in |\mathcal{A}|$, there is a functor

\[ \hat{\mathcal{A}} : \mathcal{A} \to \text{Set} \]

with $\hat{\mathcal{A}}(B) = \mathcal{A}(A, B)$ on objects. For $b \in \mathcal{A}(B, B_1)$ and $f \in \mathcal{A}(A, B)$, $\hat{\mathcal{A}}(b)(f) = b \circ f$.

\[
\begin{array}{c}
A \xrightarrow{f} B \xrightarrow{b} B_1 \\
\end{array}
\]

*Because: on morphisms:*

\[ \hat{\mathcal{A}}(b_1 \circ b)(f) = (b_1 \circ b)(f) = b_1 \circ (b \circ f) = \hat{\mathcal{A}}(b_1)(b \circ f) = \hat{\mathcal{A}}(b_1) \circ \hat{\mathcal{A}}(b)(f). \]

Such are called **hom** functors.
2.35 Dual Functors. The dual of a functor \( F : \mathcal{A} \to \mathcal{B} \) is the functor \( F^\circ : \mathcal{A}^\circ \to \mathcal{B}^\circ \) with \( F^\circ(A^\circ) = F(A) \) on objects and \( F^\circ(a^\circ) = F(a)^\circ \) on morphisms.

2.351 The dual of a functor is a functor. Because: \( F^\circ \) is well-defined on objects. It is also well-defined on morphisms: \( A_1 \xrightarrow{\alpha} A_2 \) implies \( F(A_1) \xrightarrow{F(\alpha)} F(A_2) \) implies \( F(A_1)^\circ \xrightarrow{F(\alpha)^\circ} F(A_2)^\circ \).

Moreover, \( F^\circ(id_{A^\circ}) = F^\circ((id_A)^\circ) = F(id_A)^\circ = (id_{F(A)})^\circ = id_{F(A)^\circ} \)

and

\[
F^\circ(f^\circ \circ g^\circ) = F^\circ((g \circ f)^\circ) = (F(g \circ f))^\circ = (F(g) \circ F(f))^\circ = F(f)^\circ \circ F(g)^\circ = F^\circ(f^\circ) \circ F^\circ(g^\circ).
\]

A dual construction need not reverse all arrows: the dual of \( F : \mathcal{A} \to \mathcal{B} \) is not a functor \( \mathcal{B}^\circ \to \mathcal{A}^\circ \), for example.

2.36 Composition of Functors. The composite of functors \( \mathcal{A} \xrightarrow{F} \mathcal{B} \) and \( \mathcal{B} \xrightarrow{G} \mathcal{C} \), denoted \( \mathcal{A} \xrightarrow{G \circ F} \mathcal{C} \), maps \( A \in |\mathcal{A}| \) to \( G(F(A)) \) and \( a \in \mathcal{A}(A, A') \) to \( G(F(a)) \).

2.361 Functors \( \mathcal{A} \xleftarrow{G} \mathcal{B} \) are equal if \( F(A) = G(A) \) for each \( A \in |\mathcal{A}| \) and \( F(a) = G(a) \) for each \( A \in |\mathcal{A}| \).

2.362 If \( \mathcal{A} \xrightarrow{F} \mathcal{B} \), \( \mathcal{B} \xrightarrow{G} \mathcal{C} \), and \( \mathcal{C} \xrightarrow{H} \mathcal{D} \) are functors, then \((H \circ G) \circ F = H \circ (G \circ F), F \circ id_A = F,\) and \( id_B \circ G = G \).

Because: these follow from the definition of \( \circ \) and \( = \) for functors.

2.363 If \( F : \mathcal{A} \to \mathcal{B} \) and \( G : \mathcal{B} \to \mathcal{C} \) are functors then \((G \circ F)^\circ : \mathcal{A}^\circ \to \mathcal{B}^\circ \) satisfies \((G \circ F)^\circ = G^\circ \circ F^\circ \).

Because: on objects:

\[
(G \circ F)^\circ(A^\circ) = ((G \circ F)(A))^\circ = (G(F(A)))^\circ = G^\circ(F^\circ(A^\circ)) = (G^\circ \circ F^\circ)(A^\circ)
\]

and on morphisms:

\[
(G \circ F)^\circ(f^\circ) = ((G \circ F)(f))^\circ G(F(f))^\circ = G^\circ(F^\circ(f^\circ)) = G^\circ \circ F^\circ(f^\circ).
\]

2.37 Faithful Functors. Let \( \mathcal{C} \) and \( \mathcal{D} \) be categories. A set of functors \( \{ F_\alpha : \mathcal{C} \to \mathcal{D} \mid \alpha \} \) is faithful iff given morphisms \( k \) and \( k' \) of \( \mathcal{C} \), the equations \( \Box f = \Box g, f \Box = g \Box, \) and \( F_\alpha(k) = F_\alpha(k') \) for each \( \alpha \) together imply \( k = k' \). Usage of the term faithful here is consistent with that in [87] and [102] but conflicts with that in [69].
2.371 If \( \{ F_\alpha : B \to C \mid \alpha \} \) and \( \{ G_\beta : C \to D \mid \beta \} \) are faithful then so is \( \{ G_\beta \circ F_\alpha \mid \alpha, \beta \} \).

Because: the equations \( \Box \alpha = \Box k', k\Box = k'\Box \), and \( G_\beta(F_\alpha(k)) = G_\beta(F_\alpha(k')) \) for all \( \alpha \) and \( \beta \) imply \( F_\alpha(k) = F_\alpha(k) \) for all \( \alpha \) which implies \( k = k' \).

2.372 If \( \{ F_\alpha : B \to C \mid \alpha \} \) and \( \{ G_\beta : C \to D \mid \beta \} \) are sets of functors with \( \{ G_\beta \circ F_\alpha \mid \alpha, \beta \} \) faithful then \( \{ F_\alpha : B \to C \mid \alpha \} \) is faithful.

Because: the equations \( \Box \alpha = \Box k', k\Box = k'\Box \), and \( F_\alpha(k) = F_\alpha(k') \) for all \( \alpha \) imply \( G_\beta(F_\alpha(k)) = G_\beta(F_\alpha(k)) \) for all \( \alpha \) and \( \beta \) which implies \( k = k' \).

2.373 Let \( A \) and \( B \) be categories with \( A \) founded on \( B \). The underlying functor \( U : A \to B \) is faithful.

Because: \( U \) was defined in 2.34. See [69] or [87].

2.38 Reflection of Monomorphisms. Let \( C \) and \( D \) be categories. A set of functors \( \{ F_\alpha : C \to D \mid \alpha \} \) reflects monics if given a morphism \( k \) of \( C \), \( F_\alpha(k) \) monic in \( D \) for each \( \alpha \) implies \( k \) monic.

2.381 A faithful set of functors reflects monics.

Because: for \( k \in C(C', C'') \) with each \( F_\alpha(k) \) monic,

\[
\begin{array}{ccc}
C & \xrightarrow{k} & C'' \\
\alpha \downarrow & & \downarrow \\
\alpha & \xrightarrow{x} & \alpha \circ x
\end{array}
\]

implies \( F_\alpha(k \circ x) = F_\alpha(k \circ y) \) for each \( \alpha \) implies \( F_\alpha(k) \circ F_\alpha(x) = F_\alpha(k) \circ F_\alpha(y) \) for each \( \alpha \) implies \( F_\alpha(x) = F_\alpha(y) \) for each \( \alpha \) implies \( x = y \).

2.382 Let \( C \) and \( D \) be categories. A set of functors \( \{ F_\alpha : C \to D \mid \alpha \} \) reflects epics if given a morphism \( k \) of \( C \), \( F_\alpha(k) \) epic in \( D \) for each \( \alpha \) implies \( k \) epic.

2.383 A faithful set of functors reflects epics.

Because: for \( k \in C(C, C') \) with each \( F_\alpha(k) \) epic,

\[
\begin{array}{ccc}
C & \xrightarrow{k} & C' \\
\alpha \downarrow & & \downarrow \\
\alpha & \xrightarrow{x} & \alpha \circ k
\end{array}
\]

implies \( F_\alpha(x \circ k) = F_\alpha(y \circ k) \) for each \( \alpha \) implies \( F_\alpha(x) \circ F_\alpha(k) = F_\alpha(y) \circ F_\alpha(k) \) for each \( \alpha \) implies \( F_\alpha(x) = F_\alpha(y) \) for each \( \alpha \) implies \( x = y \).

2.39 Full Functors. Let \( C \) and \( D \) be categories. A set of functors \( \{ F_\alpha : C \to D \mid \alpha \} \) is full if for each morphism \( d \) of \( D \), \( d = F_\alpha(k) \) for some \( \alpha \) and some morphism \( k \) of \( C \).
2.3. FUNCTORS 17

2.391 A subcategory is full iff its inclusion functor is full.

2.3a Reflection of Isomorphisms. Let \( C \) and \( D \) be categories. A set of functors \( \{ F_\alpha : C \to D \mid \alpha \} \) reflects isomorphisms if \( F_\alpha(a) \) an isomorphism for each \( \alpha \) implies \( a \) an isomorphism.

2.3b Conservative Functors. Let \( C \) and \( D \) be categories. A set of functors \( \{ F_\alpha : C \to D \mid \alpha \} \) is conservative if it is faithful and reflects isomorphisms. This term is from [87]. In [69], the term faithful is used.

2.3b1 If \( C \) is balanced then \( \{ F_\alpha : C \to D \mid \alpha \} \) is conservative iff it is faithful.

Because: see [87].

2.3b2 A full and faithful set, \( \{ F_\alpha : C \to D \mid \alpha \} \), of functors is conservative.

2.3c Separators in a Category. A set \( \mathcal{G} \) of objects in a category \( C \) is separating if for any morphisms \( X \xrightarrow{f} Y \), if \( f \neq g \), then there exists \( G \in \mathcal{G} \) and \( u \in C(G, A) \) for which \( f \circ u \neq g \circ u \). If \( \mathcal{G} = \{ G \} \) consists of a single object \( G \), then \( G \) is a separator.

In [102] MacLane uses the term generating for such a collection of objects but indicates that separating is better. The term generating is also used in [69]. Johnstone uses separating in [87].

2.3c1 A set \( \mathcal{G} \) of objects in a category \( C \) is separating iff the set \( \{ \tilde{G} : C \to \text{Set} \mid G \in \mathcal{G} \} \) of functors is faithful.

Because: see pages 12–13 in Volume I of [87].

2.3d Generators in a Category. A set \( \mathcal{G} \) of objects in a category \( C \) is generating if it is separating and if, given \( f \in C(A, B) \), the conditions

i) for any \( G \in \mathcal{G} \) and \( v \in C(G, B) \), there exists \( u \in C(G, A) \) with \( v = f \circ u \);

ii) for any \( G \in \mathcal{G} \), if \( u, u' \in C(G, A) \) and \( f \circ u = f \circ u' \), then \( u = u' \);

imply that \( f \) is an isomorphism. If \( \mathcal{G} = \{ G \} \) consists of a single object \( G \), then \( G \) is a generator.

The term basis is used in [69] for such a set of objects.

2.3d1 A set \( \mathcal{G} \) of objects in a category \( C \) is generating iff the set \( \{ \tilde{G} : C \to \text{Set} \mid G \in \mathcal{G} \} \) of functors is conservative.

Because: see pages 12–13 in Volume I of [87].
2.4 Natural transformations

Let \( B \) and \( C \) be categories and let \( F, G : B \to C \) be functors. A class of morphisms

\[
\tau = \{ \tau_B \in C(F(B), G(B)) \mid B \in |B| \}
\]

is a transformation, denoted \( \tau : F \to G : B \to C \). Each \( \tau_B \) is a component. Transformations are equal if their components are.

2.4.1 Transformations. A transformation \( \tau : F \to G : B \to C \) is **natural** if

\[
\begin{array}{ccc}
F(B) & \xrightarrow{\tau_B} & G(B) \\
F(b) & & F(b) \\
F(B') & \xrightarrow{\tau_{B'}} & G(B')
\end{array}
\]

when \( B, B' \in |B| \) and \( b \in B(B, B') \). \( \tau : F \Rightarrow G : B \to C \) indicates that \( \tau \) is a natural transformation from a functor \( F : B \to C \) to a functor \( G : B \to C \).

2.4.11 Given \( F : B \to C \), there is a natural transformation \( \text{id}_F : F \Rightarrow F : B \to C \) with

\[
(id_F)_B = id_{F(B)}.
\]

Because: \( b \in B(B, B') \) implies

\[
\begin{array}{ccc}
F(B) & \xrightarrow{id_F(B)} & F(B) \\
F(b) & & F(b) \\
F(B') & \xrightarrow{id_F(B')} & F(B')
\end{array}
\]

2.42 Dual Natural Transformations. The **dual** of a natural transformation \( \alpha : F \Rightarrow G : A \to B \) is the natural transformation

\[
\alpha^\circ : G^\circ \Rightarrow F^\circ : A^\circ \to B^\circ
\]

with \( \alpha^\circ_{A^\circ} = (\alpha_A)^\circ \).

2.4.21 The dual of a natural transformation is a natural transformation.

Because: \( A_1^\circ \xrightarrow{a^\circ} A_2^\circ \) implies \( A_1 \xleftarrow{a} A_2 \) implies

\[
\begin{array}{ccc}
F(A_2) & \xrightarrow{\alpha_{A_2}} & G(A_2) \\
F(a) & & F(a) \\
F(A_1) & \xrightarrow{\alpha_{A_1}} & G(A_1)
\end{array}
\]
implies

\[
\begin{align*}
F^\circ(A_2^2) & \xrightarrow{\alpha_{A_2^2}} G^\circ(A_2^2) \\
F^\circ(a^\circ) & \downarrow \quad \downarrow G^\circ(a^\circ) \\
F^\circ(A_1^2) & \xleftarrow{\alpha_{A_1^2}} G^\circ(A_1^2)
\end{align*}
\]

which is

\[
\begin{align*}
G^\circ(A_1^2) & \xrightarrow{\alpha_{A_1^2}} F^\circ(A_1^2) \\
G^\circ(a^\circ) & \downarrow \quad \downarrow F^\circ(a^\circ) \\
G^\circ(A_2^2) & \xleftarrow{\alpha_{A_2^2}} F^\circ(A_2^2)
\end{align*}
\]

Again, the dual construction does not reverse all arrows: the dual of \( \alpha : F \Rightarrow G : A \to B \) is not a natural transformation \( G^\circ \Rightarrow F^\circ : B^\circ \to A^\circ \).

### 2.43 Composition of Natural Transformations.

For \( \alpha : F \Rightarrow F' : A \to B \) and \( \beta : F' \Rightarrow F'' : A \to B \) as shown in

\[
\begin{align*}
& \xrightarrow{\alpha} \\
& \xrightarrow{\beta} \\
\end{align*}
\]

\( \beta \circ \alpha : F \Rightarrow F'' : A \to B \) with \( (\beta \circ \alpha)_A = \beta_A \circ \alpha_A \) is the composite of \( \alpha \) and \( \beta \). Some texts, notably [102], use \( \cdot \) to denote a composite of natural transformations.

#### 2.431 \( \beta \circ \alpha \) is a natural transformation.

Because: \( a \in A(A, A') \) implies

\[
\begin{align*}
F(A) & \xrightarrow{\alpha_A} F'(A) \xrightarrow{\beta_A} F''(A) \\
F(a) & \downarrow \quad \downarrow F'(a) \\
F(A') & \xrightarrow{\alpha_{A'}} F'(A') \xrightarrow{\beta_{A'}} F''(A')
\end{align*}
\]

#### 2.432 \( \alpha : F \Rightarrow G : B \to C, \beta : G \Rightarrow H : B \to C, \) and \( \gamma : H \Rightarrow K : B \to C \) natural transformations implies

\( (\gamma \circ \beta) \circ \alpha = \gamma \circ (\beta \circ \alpha) \).

#### 2.433 \( \alpha : F \Rightarrow G : B \to C \) a natural transformation implies

\( \alpha \circ id_F = \alpha = id_G \circ \alpha \).
2.434 $\alpha : F \Rightarrow G : A \rightarrow B$ and $\beta : G \Rightarrow H : A \rightarrow B$ induce $\beta^\circ : H^\circ \Rightarrow G^\circ : A^\circ \rightarrow B^\circ$ and $\alpha^\circ : G^\circ \Rightarrow F^\circ : A^\circ \rightarrow B^\circ$. Moreover, $(\alpha^\circ \circ \beta^\circ)^A = \alpha^A_\circ \circ \beta^A_\circ$ by definition of composites of natural transformations and this equals $(\beta_\circ \circ \alpha_\circ)^A$.

2.44 Products of Natural Transformations. For $\alpha : F \Rightarrow F' : A \rightarrow B$ and $\beta : G \Rightarrow G' : B \rightarrow C$ as shown in

$$
\begin{array}{ccc}
A & \xrightarrow{\alpha} & B \\
\downarrow F & & \downarrow G \\
F'(A') & \xrightarrow{\beta} & G'(A')
\end{array}
$$

$\beta \circ \alpha : G \circ F \Rightarrow G' \circ F' : A \rightarrow C$ defined by $(\beta \circ \alpha)_A = \beta_{F'(A)} \circ G(\alpha_A)$ is the product of $\alpha$ and $\beta$. Some texts, notably [102], use $\circ$ to denote a product of natural transformations.

2.441 $\beta \circ \alpha$ is a well-defined natural transformation. Moreover, $(\beta \circ \alpha)_A = G'(\alpha_A) \circ \beta_{F(A)}$.

Because: $a \in A(A, A')$ implies

$$
\begin{array}{ccc}
F(A) & \xrightarrow{\alpha_A} & F'(A) \\
\downarrow F(a) & & \downarrow F'(a) \\
F'(A') & \xrightarrow{\alpha_{A'}} & F'(A')
\end{array}
$$

which implies

$$
\begin{array}{ccc}
G(F(A)) & \xrightarrow{G(\alpha_A)} & G(F'(A)) \\
\downarrow G(F(a)) & & \downarrow G(F'(a)) \\
G(F'(A')) & \xrightarrow{G(\alpha_{A'})} & G(F'(A'))
\end{array}
$$

Naturality of $\beta$ gives

$$
\begin{array}{ccc}
G(F'(A)) & \xrightarrow{G(F'(a))} & G(F'(A')) \\
\downarrow G(F'(a)) & & \downarrow G(F'(a)) \\
G(F'(A')) & \xrightarrow{G(F'(a'))} & G(F'(A'))
\end{array}
$$

and together these imply naturality of $\beta \circ \alpha$.

Naturality of $\beta$ justifies the alternative formula: $\alpha_A \in B(F(A), F'(A))$ so

$$
\begin{array}{ccc}
G(F(A)) & \xrightarrow{G(\alpha_A)} & G(F'(A)) \\
\downarrow G(F(\alpha)) & & \downarrow G(F'(\alpha)) \\
G(F'(A)) & \xrightarrow{G(F'(\alpha))} & G(F'(A'))
\end{array}
$$

2.442 $\alpha : F \Rightarrow G : B \rightarrow C$ a natural transformation implies

$$
id_{(id_C)} \circ \alpha = \alpha = \alpha \circ id_{(id_B)}$$

where the first and last terms are identity natural transformations for identity functors.
2.4. NATURAL TRANSFORMATIONS

2.443  If \( \alpha : F \Rightarrow F' \), \( \alpha' : F' \Rightarrow F'' \), \( \beta : G \Rightarrow G' \) and \( \beta' : G' \Rightarrow G'' \) are natural transformations as in

\[
\begin{array}{ccc}
A & \overset{F}{\longrightarrow} & B \\
\alpha \downarrow & & \downarrow \beta \\
F'' & \overset{G}{\longrightarrow} & C \\
\alpha' \downarrow & & \downarrow \beta' \\
B & \overset{G'}{\longrightarrow} & C
\end{array}
\]

then

\[
(\beta' \circ \beta) \ast (\alpha' \circ \alpha) = (\beta' \ast \alpha') \circ (\beta \ast \alpha).
\]

Because: for \( A \in |\mathcal{A}| \), definitions of \( \circ \) and \( \ast \) imply the first and last equalities of

\[
[(\beta' \circ \beta) \ast (\alpha' \circ \alpha)]_A = (\beta' \circ \beta)_{F''(A)} \circ G((\alpha' \circ \alpha)_A)
\]

\[
= \beta'_{F''(A)} \circ \beta_{F''(A)} \circ G(\alpha'_A) \circ G(\alpha_A)
\]

\[
= \beta'_{F''(A)} \circ G(\alpha'_A) \circ \beta_{F''(A)} \circ G(\alpha_A)
\]

\[
= (\beta' \ast \alpha'_A) \circ (\beta \ast \alpha)_A
\]

\[
= [(\beta' \ast \alpha') \circ (\beta \ast \alpha)]_A
\]

while naturality of \( \beta \) and \( \beta' \) imply equalities three and four.

2.444  \( \alpha : F_1 \Rightarrow F_2 : \mathcal{A} \rightarrow \mathcal{B} \) and \( \beta : G_1 \Rightarrow G_2 : \mathcal{B} \rightarrow \mathcal{C} \) induce \( \alpha^\circ : F_2^\circ \Rightarrow F_1^\circ : \mathcal{A}^\circ \rightarrow \mathcal{B}^\circ \) and \( \beta^\circ : G_2^\circ \Rightarrow G_1^\circ : \mathcal{B}^\circ \rightarrow \mathcal{C}^\circ \). Moreover, \( (\beta^\circ \ast \alpha^\circ)_A = \beta^\circ_{F_2^\circ(A^\circ)} \circ \alpha^\circ_{F_1^\circ(A^\circ)} \) by definition of products of natural transformations and this equals \( (G_2(\alpha_A) \circ \beta_{F_1(A)})^\circ = ((\beta \ast \alpha)_A)^\circ \).

2.45  Natural Isomorphisms. \( \tau : F \Rightarrow G : \mathcal{B} \rightarrow \mathcal{C} \) is a natural isomorphism if there exists \( \gamma : G \Rightarrow F : \mathcal{B} \rightarrow \mathcal{C} \) for which \( \tau \circ \gamma = \text{id}_G \) and \( \gamma \circ \tau = \text{id}_F \).

2.451  \( \tau : F \Rightarrow G : \mathcal{B} \rightarrow \mathcal{C} \) is a natural isomorphism iff each component \( \tau_B \) is an isomorphism in \( \mathcal{C} \).

Because: evaluate \( \tau \circ \tau^{-1} = \text{id}_G \) and \( \tau^{-1} \circ \tau = \text{id}_F \) at each component.

2.452  If \( \mathcal{A} \) is a locally small category and \( A \in |\mathcal{A}| \), there is a natural isomorphism

\[
\delta_A : \hat{\mathcal{A}}^\circ \Rightarrow \hat{\mathcal{A}} : \mathcal{A}^\circ \rightarrow \textbf{Set}
\]

with \((\delta_A)_B^\circ : \mathcal{A}^\circ(\mathcal{A}^\circ, B^\circ) \rightarrow \mathcal{A}(A, B)\) the bijection \( b^\circ \mapsto b \).

Because: \( B^\circ \in |\mathcal{A}^\circ| \) implies \((\delta_A)_B^\circ : \mathcal{A}^\circ(\mathcal{A}^\circ, B^\circ) \rightarrow \mathcal{A}(A, B)\) well-defined. Hence, \( \delta_A : \hat{\mathcal{A}}^\circ \rightarrow \hat{\mathcal{A}} : \mathcal{A}^\circ \rightarrow \textbf{Set} \) is a well-defined transformation. For \( b^\circ \in \mathcal{A}^\circ(B^\circ, B'_1) \), commutativity of

\[
\begin{array}{ccc}
\mathcal{A}^\circ(A^\circ, B^\circ) & \overset{(\delta_A)_B^\circ}{\longrightarrow} & \mathcal{A}(B, A) \\
\hat{\mathcal{A}}^\circ(b^\circ) \downarrow & & \downarrow \hat{\mathcal{A}}(b^\circ) \\
\mathcal{A}^\circ(A^\circ, B'_1) & \overset{(\delta_A)_B^\circ}{\longrightarrow} & \mathcal{A}(B_1, A)
\end{array}
\]
is
\[(\delta_A)_{B_1^0}(\tilde{A}^0(b^0)(f^0)) = (\delta_A)_{B_1^0}(b^0 \circ f^0) = (\delta_A)_{B_1^0}((f \circ b)^0) = f \circ b = \hat{A}(b^0)(f).\]

The notation \(\tilde{A}^0\) in the last claim refers to the functor \(\mathcal{A}^0 \to \text{Set}\) obtained from \(\mathcal{A}^0 \in \mathcal{A}^0\) by application of 2.344. It should not be confused with \((\hat{A})^0\), which is a functor \(\mathcal{A}^0 \to \text{Set}^\circ\).

2.453 If \(\mathcal{A}\) is a locally small category then \(a \in \mathcal{A}(A_1, A)\) induces a natural transformation
\[\hat{a} : \hat{A}_1 \Rightarrow \hat{A} : \mathcal{A}^0 \to \text{Set}.\]

Its component at \(B^0 \in |\mathcal{A}|\) is the function \(\hat{a}_B^0 : \mathcal{A}(B, A_1) \to \mathcal{A}(B, A)\) defined by \(\hat{a}_B^0(f) = a \circ f\) for \(B \in |\mathcal{A}|\) and \(f \in \mathcal{A}(B, A_1)\).

Because: the diagram
\[
\begin{array}{ccc}
B & \xrightarrow{f} & A_1 \\
\downarrow \hat{a} & & \downarrow \hat{A}_1 \\
A & \xrightarrow{\hat{a}_B^0} & A
\end{array}
\]

indicates that components of \(\hat{a}\) are well-defined functions. Naturality is commutativity of
\[
\begin{array}{ccc}
\mathcal{A}(B_1, A_1) & \xrightarrow{\hat{a}_B^0} & \mathcal{A}(B, A) \\
\downarrow \hat{A}_1(b^0) & & \downarrow \hat{A}(b^0) \\
\mathcal{A}(B, A_1) & \xrightarrow{\hat{a}_B^0} & \mathcal{A}(B, A)
\end{array}
\]

for each \(b^0 \in \mathcal{A}^0(B_1^0, B^0)\) and is implied by
\[\hat{a}_B^0(\hat{A}_1(b^0)(f)) = \hat{a}_B^0(f \circ b) = a \circ (f \circ b) = (a \circ f) \circ b = \hat{A}(b^0)(a \circ f) = \hat{A}(b^0)(\hat{a}_B^0(f)).\]

2.454 If \(\mathcal{A}\) is a locally small category then \(a \in \mathcal{A}(A, A_1)\) induces a natural transformation
\[\hat{a} : \hat{A}_1 \Rightarrow \hat{A} : \mathcal{A}^0 \to \text{Set}.\]

Its component at \(B \in |\mathcal{A}|\) is the function \(\hat{a}_B : \mathcal{A}(A_1, B) \to \mathcal{A}(A, B)\) defined by \(f \mapsto f \circ a\).

Because: the previous claim gives \(\hat{a}^0\) (obtained from \(a^0\) by applying \(\hat{\cdot}\)) which is the middle natural transformation of
\[\hat{A}_1 \Rightarrow \hat{A}_1^0 \Rightarrow \hat{A}^0 \Rightarrow \hat{A}\]

while \(\delta_{A_1^0}\) and \(\delta_{A^0}\) are the first and third.

The Yoneda lemma implies that these are the only natural transformations between hom functors.

2.46 Equivalences of Categories. Let \(\mathcal{X}\) and \(\mathcal{A}\) be categories. An equivalence of from \(\mathcal{X}\) \(\mathcal{A}\) is a structure \((F, G, \eta, \varepsilon)\) with \(F : \mathcal{X} \to \mathcal{A}\) and \(G : \mathcal{A} \to \mathcal{X}\) functors and with \(\eta : \text{id}_{\mathcal{X}} \Rightarrow G \circ F\) and \(\varepsilon : F \circ G \Rightarrow \text{id}_{\mathcal{A}}\) natural isomorphisms. If \(\mathcal{C}\) and \(\mathcal{D}\) are categories, then
\[\mathcal{C} \approx \mathcal{D}\]
indicates that there is an equivalence between them.
2.461 \((F,G,\eta,\varepsilon)\) is an equivalence from \(\mathcal{X}\) to \(\mathcal{A}\) iff \((G,F,\varepsilon^{-1},\eta^{-1})\) is an equivalence from \(\mathcal{A}\) to \(\mathcal{X}\).

*Because:* this follows from the definition of equivalence of categories.

2.462 Let \(\mathcal{C}\) be a category and let \(F : \mathcal{C} \to \textbf{Set}\) be a functor. There is a category \(\mathcal{F}\) such that \(\text{Set}^{\mathcal{C}}/F \approx \text{Set}^F\).

Moreover, if \(\mathcal{C}\) is small, then \(\mathcal{F}\) is as well.

*Because:* this is A1.1.7 from [87].

Categories of \textbf{Set}-valued functors have useful properties: they are complete, cocomplete, and toposes. The above claim asserts that every slice of such a category is equivalent to another such category, hence, inherits the useful properties.

2.463 If \(B\) is a set, then there is an equivalence \(\text{Set}/B \approx \text{Set}^B\).

*Because:* this is a consequence of 2.462 obtained using \(\mathcal{C} = 1\) and \(\mathcal{F} = B\) (viewed as a discrete category).

2.47 **Functor Categories.** For \(\mathcal{C}\) a category and \(\mathcal{J}\) a small category, \(\mathcal{C}^\mathcal{J}\) is the category having functors \(\mathcal{J} \to \mathcal{C}\) as objects and having natural transformations between such functors as morphisms. \(\mathcal{C}^\mathcal{J}\) is a (locally small) category.

*Because:* \(F, G : \mathcal{J} \to \mathcal{C}\) implies

\[
\mathcal{C}^\mathcal{J}(F, G) \subset \bigcup_{J \in |\mathcal{J}|} \mathcal{C}(F(J), G(J)).
\]

2.411 and 2.43 respectively give identities and composites.

2.471 For a category \(\mathcal{C}\) and a small category \(\mathcal{J}\), the diagonal functor \(\Delta : \mathcal{C} \to \mathcal{C}^\mathcal{J}\) has \(\Delta(C)(J) = C\) and \(\Delta(C)(j) = id_C\) on objects and \(\Delta(k)_{J} = k\) on morphisms.

If \(\mathcal{J}\) has a single object and only the identity morphism then \(\mathcal{C}^\mathcal{J} \cong \mathcal{C}\). The following fixes notation for a particular such \(\mathcal{J}\):

\[
1 = \ast \bigcup id_\ast.
\]

2.472 For a category \(\mathcal{C}\) and a small category \(\mathcal{J}\), there is an evaluation functor \(ev : \mathcal{C}^\mathcal{J} \times \mathcal{J} \to \mathcal{C}\) via \(ev(F, J) = F(J)\) and \(ev(\tau, j) = F'(j) \circ \tau_J = \tau_{J'} \circ F(j)\) for \(\tau : F \Rightarrow F'\) and \(j \in \mathcal{J}(J, J')\). Fixing \(J \in |\mathcal{J}|\) gives \(ev(\cdot, J)\) denoted \(ev_J : \mathcal{C}^\mathcal{J} \to \mathcal{C}\).

2.473 \(\mathcal{J}\) a small category implies \(\{ ev_J : \mathcal{C}^\mathcal{J} \to \mathcal{C} \mid J \in |\mathcal{J}| \}\) is faithful.

*Because:* equality of natural transformations is defined as equality of all components.
2.474 \( J \) a small category implies \( \{ ev_J : C^J \to C \mid J \in |J| \} \) reflects monics and epics.

Because: it is faithful 2.473, 2.381, 2.383.

2.475 Let \( C \) and \( D \) be categories with \( D \) small and discrete. A morphism \( \tau \) of \( C^D \) is monic iff each of its components is monic in \( C \). Similarly, \( \tau \) is epic iff each component is.

Because: \( \Leftrightarrow \{ ev_D \mid D \in |D| \} \) is faithful so reflects monics and epics 2.381, 2.383.

\( \Rightarrow \) Given \( \tau : G \Rightarrow G' : D \to C \) monic, fix \( \overline{D} \in |D| \) and assume
\[
C \xrightarrow{\tau}(G(\overline{D})) \xrightarrow{\gamma} G'(\overline{D}).
\]

Define \( \overline{G} : D \to C \) by
\[
\overline{G}(D) = \begin{cases} C & \text{if } D = \overline{D}; \\ G(D) & \text{otherwise} \end{cases}
\]
on objects and \( \overline{G}(id_D) = id_{\overline{G}(D)} \) on morphisms. Define \( \hat{x} : \overline{G} \Rightarrow G \) and \( \hat{y} : G \Rightarrow \overline{G} \) by
\[
\hat{x}_D = \begin{cases} x & \text{if } D = \overline{D}; \\ id_{\overline{G}(D)} & \text{otherwise} \end{cases} \quad \text{and} \quad \hat{y}_D = \begin{cases} y & \text{if } D = \overline{D}; \\ id_{\overline{G}(D)} & \text{otherwise} \end{cases}
\]
These are natural since \( D \) is discrete. \( D \in |D| \) implies \( (\tau \circ \hat{x})_D = (\tau \circ \hat{y})_D \) implies \( \tau \circ \hat{x} = \tau \circ \hat{y} \) implies \( \hat{x} = \hat{y} \) implies \( x = y \). Thus, \( \overline{\tau} \) is monic in \( C \).

2.476 Let \( A \) and \( B \) be small categories and let \( C \) be a category. A functor \( F : A \to B \) induces \( C^F : C^B \to C^A \) via \( X \mapsto X \circ F \) and \( \tau \mapsto \tau \circ id_F \).

2.477 If a small functor \( F : A \to B \) is surjective on objects then \( C^F : C^B \to C^A \) is conservative.

Because: \( C^F(\tau) = C^F(\tau') \) and \( B \in |B| \) implies there is \( A_B \in |A| \) for which \( F(A_B) = B \). Then
\[
C^F(\tau)_{A_B} = C^F(\tau')_{A_B} \\
(\tau \circ id_F)_{A_B} = (\tau' \circ id_F)_{A_B} \\
\tau_{F(A_B)} = \tau'_{F(A_B)} \\
\tau_B = \tau'_B
\]
implies \( \tau = \tau' \), hence, \( C^F \) is faithful.

\( \tau : G \Rightarrow G' : B \to C \) with \( C^F(\tau) : G \circ F \Rightarrow G' \circ F : A \to C \) a natural isomorphism gives
\[
\begin{align*}
G \circ F \xrightarrow{\tau \circ id_F} G' \circ F \\
\downarrow id_{G \circ F} & \quad \downarrow \gamma \\
G \circ F \xrightarrow{\tau \circ id_F} G' \circ F
\end{align*}
\]
for some \( \gamma : G' \circ F \Rightarrow G \circ F : A \to C \). Given \( B \in |B| \), there is \( A_B \in |A| \) for which \( F(A_B) = B \). This gives
\[
\gamma_{A_B} \circ \tau_B = id_{G(B)} \quad \text{and} \quad \tau_B \circ \gamma_{A_B} = id_{G'(B)}
\]
hence, $\tau_B$ is an isomorphism in $C$. $\tau$ is an isomorphism since each of its components is.

2.478 If $F : A \to B$ is a functor and $J$ is a small category then there is a functor $F^J : A^J \to B^J$ defined by $X \mapsto F \circ X$ and $\tau \mapsto id_F \ast \tau$.

2.48 Comma Categories. For a functor $F : A \to B$ and an object $B \in |B|$, $F \downarrow B$ is the category founded on $A$ having as objects pairs $(A, \beta)$ with $A \in |A|$ and $\beta \in B(F(A), B)$. Its source-target predicate is given by $(A, \beta) \xrightarrow{a} (A', \beta')$ iff $\Box a = A$, $a \Box = A'$, and $\beta' \circ F(a) = \beta$ as depicted in

\[
\begin{array}{c}
\xymatrix{F(A) \ar[r]^-{F(a)} \ar[d]_\beta & F(A') \ar[d]_-{\beta'} \\
B} \\
\end{array}
\]

This and the the construction in 2.482 are special cases of comma categories. See page 46 of [102]. If $F = id_A$, then $F \downarrow A$ is denoted $C/A$.

2.481 Let $F : A \to B$ be a small functor and let $C$ be a category. For each $B \in |B|$ there is a conservative functor $B \downarrow F \xrightarrow{B_F} A$ via $(A, b) \mapsto A$ and $a \mapsto a$.

Because: $a : (A, \beta) \to (A', \beta')$ with

\[
\begin{array}{c}
\xymatrix{A \ar[r]^a \ar[d]_{id_A} & A' \ar[d]^{a'} \\
A} \\
\end{array}
\]

and

\[
\begin{array}{c}
\xymatrix{A' \ar[r]^-{a'} \ar[d]_{id_{A'}} & A \ar[d]^-{a} \\
A'} \\
\end{array}
\]

implies

\[
\begin{align*}
\beta' \circ F(a) &= \beta \\
\beta' \circ F(a) \circ F(a') &= \beta \circ F(a') \\
\beta' \circ F(a \circ a') &= \beta \circ F(a') \\
\beta' &= \beta \circ F(a')
\end{align*}
\]

implies $a' : (A', \beta') \to (A, \beta)$ an inverse of $a$ in $F \downarrow b$.

2.482 For a functor $F : A \to B$ and an object $B \in |B|$, $B \downarrow F$ is the category founded on $A$ having as objects pairs $(A, \beta)$ with $A \in |A|$ and $\beta \in B(B, F(A))$. Its source-target predicate is given by $(A, \beta) \xrightarrow{a} (A', \beta')$ iff $\Box a = A$, $a \Box = A'$, and $F(a) \circ \beta = \beta'$ as depicted in

\[
\begin{array}{c}
\xymatrix{B \ar[r]^\beta \ar[d]_{\beta'} & F(A') \ar[d]_-{F(a)} \\
F(A) \ar[r]_-{F(a)} & F(A')} \\
\end{array}
\]

If $F = id_A$, then $A \downarrow F$ is denoted $A \setminus A$. 
2.483 \( F : A \to B \) a functor and \( B \in |B| \) implies

\[
(F \downarrow B)^o \cong B^o \downarrow F^o \quad \text{and} \quad (B \downarrow F)^o \cong F^o \downarrow B^o.
\]

Because: the former is illustrated as follows. \((A, \beta)^o \overset{a^o}{\longrightarrow} (A_1, \beta_1)^o\) in \((F \downarrow B)^o\) gives \((A_1, \beta_1) \overset{a}{\longrightarrow} (A, \beta)\) in \((F \downarrow B)\) gives

\[
\begin{array}{ccc}
F(A_1) & \overset{F(a)}{\longrightarrow} & F(A) \\
\downarrow_{\beta_1} & & \downarrow_{\beta} \\
B & & B
\end{array}
\]

in \(B\) which gives

\[
\begin{array}{ccc}
F(A_1)^o & \overset{F(a)^o}{\longleftarrow} & F(A)^o \\
\downarrow_{\beta_1^o} & & \downarrow_{\beta^o} \\
B^o & & B^o
\end{array}
\]

in \(B^o\) which is

\[
\begin{array}{ccc}
F^o(A_1^o) & \overset{F^o(a^o)}{\longleftarrow} & F^o(A^o) \\
\downarrow_{\beta_1^o} & & \downarrow_{\beta^o} \\
B^o & & B^o
\end{array}
\]

and gives \((A^o, \beta^o) \overset{a^o}{\longrightarrow} (A_1^o, \beta_1^o)\) in \(B^o \downarrow F^o\). This is an isomorphism of categories. The other result is similar.

\[\square\]

2.484 Let \( F : A \to B \) be a small functor and let \( C \) be a category. For each \( B \in |B| \) there is a conservative functor \( F_B : F \downarrow B \to A \) via \((A, b) \mapsto A\) and \( a \mapsto a\).

Because: a functor is conservative iff its dual is. \( B^o \downarrow F^o \overset{B^o_{F^o}}{\longrightarrow} A^o \) gives \((F \downarrow B)^o \overset{\cong}{\longrightarrow} B^o \downarrow F^o \overset{B^o_{F^o}}{\longrightarrow} A^o\) implies \( F \downarrow B \overset{F^o_B}{\longrightarrow} A\).

\[\square\]

2.49 Yoneda Lemma. Let \( A \) be a small category and let \( F : A \to \text{Set} \) be a functor. For any \( A \in |A| \) there is a bijection \( y_{A,F} : \text{Set}^A(\hat{A}, F) \to F(A) \) via

\[
y_{A,F}(\tau) = \tau_A(id_A).
\]

Its inverse \( y_{A,F}^{-1} : F(A) \to \text{Set}^A(\hat{A}, F) \) is defined for \( x \in F(A) \), \( B \in |A| \), and \( f \in A(A, B) \) by

\[
y_{A,F}^{-1}(x)_B(f) = F(f)(x).
\]

Because: \( \tau : \hat{A} \Rightarrow F : A \to \text{Set} \) a natural transformation and \( A \in |A| \) implies \( \tau_A : A(A, A) \to F(A) \) a function. This implies \( \tau_A(id_A) \in F(A) \) and that \( y_{A,F} \) is a well-defined function.
2.4. NATURAL TRANSFORMATIONS

For \( x \in F(A) \), \( y_{A,F}^{-1}(x) \) is to be a natural transformation \( \hat{A} \Rightarrow F \). A component \( y_{A,F}^{-1}(x)_B \) for \( B \in |A| \) is to be a function \( A(A, B) \rightarrow F(B) \). For \( f \in A(A, B) \), \( y_{A,F}^{-1}(x)_B(f) \) is to be in \( F(B) \). The definition

\[
y_{A,F}^{-1}(x)_B(f) = F(f)(x)
\]

implies \( y_{A,F}^{-1}(x)_B : A(A, B) \rightarrow F(B) \) is a well-defined function, hence, \( y_{A,F}^{-1}(x) : \hat{A} \rightarrow F \) is a well-defined transformation. Fix \( f \in A(B, B_1) \). The calculation

\[
F(b) \left( y_{A,F}^{-1}(x)_B(f) \right) = F(b)(F(f)(x)) = F(b \circ f)(x) = F(A, b)(f)(x) = y_{A,F}^{-1}(x)_{B_1}(A, b)(f)
\]

implies commutativity of

\[
A(A, B) \xrightarrow{y_{A,F}^{-1}(x)_B} F(B) ,
\]

\[
A(A, B) \xrightarrow{y_{A,F}^{-1}(x)_{B_1}} F(B_1)
\]

and gives naturality of \( y_{A,F}^{-1}(x) \). It follows that \( y_{A,F}^{-1} \) is a well-defined function.

To see that \( y_{A,F} \) and \( y_{A,F}^{-1} \) are inverses, first fix \( x \in F(A) \).

\[
y_{A,F}(y_{A,F}^{-1}(x)) = y_{A,F}^{-1}(x)_A(id_A) = F(id_A)(x) = id_{F(A)}(x) = x
\]

implies \( y_{A,F} \circ y_{A,F}^{-1} = id_{F(A)} \). For \( \tau : \hat{A} \Rightarrow F : A \rightarrow \mathbf{Set} \), naturality of \( \tau \) implies equality three of

\[
y_{A,F}^{-1}(y_{A,F}^{-1}(\tau)_B(f)) = F(f)(y_{A,F}^{-1}(\tau)) = F(f)(\tau_A(id_A)) = \tau_B(A, f)(id_A) = \tau_B(f).
\]

\[
y_{A,F}^{-1}(y_{A,F}(\tau)) = \tau_B \text{ for any } B \in |A|, \text{ implies } y_{A,F}^{-1}(y_{A,F}(\tau)) = \tau \text{ and } y_{A,F}^{-1} \circ y_{A,F} = id_{\mathbf{Set}^A(\hat{A}, F)}.
\]

2.491 \( y_{A,\hat{A}_1} : \mathbf{Set}^A(\hat{A}, \hat{A}_1) \rightarrow A(A_1, A) \) via \( y_{A,\hat{A}_1}(\tau) = \tau_A(id_A) \) is a bijection with inverse

\[
y_{A,\hat{A}_1}^{-1}(a)_B(f) = \hat{A}_1(f)(a) = f \circ a
\]

for \( B \in |A| \) and \( f \in A(A, B) \).

2.492 Let \( A \) be a small category and let \( F : A^o \rightarrow \mathbf{Set} \) be a functor. For any \( A \in |A| \) there is a bijection

\[
j_{A,F} : \mathbf{Set}^{A^o}(\hat{A}, F) \rightarrow F(A^o)
\]

via \( j_{A,F}(\tau) = \tau_A(id_A) \). Its inverse

\[
j_{A,F}^{-1} : F(A^o) \rightarrow \mathbf{Set}^{A^o}(\hat{A}, F)
\]

is defined for \( x \in F(A^o) \), \( B^o \in |A^o| \), and \( f \in A(B, A) \) by \( j_{A,F}^{-1}(x)_{B^o}(f) = F(f)(x) \).

Because: by 2.452,

\[
\mathbf{Set}^{A^o}(\hat{A}, F) \xrightarrow{\delta_{A,F}} \mathbf{Set}^{A^o}(\hat{A}, F)
\]
defined by \( \tau \mapsto \tau \circ \delta_A \) is a bijection:

\[
\tilde{A}^\circ \xrightarrow{\delta_A} \tilde{A} \xrightarrow{\tau} F.
\]

2.49 provides the other bijection in

\[
\text{Set}^{A^\circ}(\tilde{A}, F) \xrightarrow{\delta_{A,F}} \text{Set}^{A^\circ}(\tilde{A}^\circ, F) \xrightarrow{y_{A^\circ,F}} F(A^\circ).
\]

Checking the formula,

\[
y_{A^\circ,F}(\delta_{A,F}(\tau)) = \delta_{A,F}(\tau)_{A^\circ}(id_{A^\circ}) = \tau_{A^\circ}((\delta_A)_{A^\circ}(id_{A^\circ})) = \tau_{A^\circ}(id_A)
\]

2.493 \( j_{A,\tilde{A}_1} : \text{Set}^{A^\circ}(\tilde{A}, \tilde{A}_1) \to \mathcal{A}(A, A_1) \) via \( j_{A,\tilde{A}_1}(\tau) = \tau_{A^\circ}(id_A) \) is a bijection with inverse

\[
j_{A,\tilde{A}_1}^{-1}(a)_{B^\circ}(f) = a \circ f
\]

for \( B \in |A^\circ| \) and \( f \in \mathcal{A}(B, A) \).

2.494 Let \( A \) be a small category and let \( F : A \to \text{Set} \) and \( G : A \to \text{Set} \) be functors. \( \tau : F \Rightarrow G \) is monic in \( \text{Set}^A \) iff each component of \( \tau \) is monic in \( A \).

Because: \( \Leftarrow \): \( \tau \circ \gamma = \tau \circ \gamma' \) implies \( \tau_A \circ \gamma_A = \tau_A \circ \gamma_A' \) for each \( A \in |A| \) implies \( \gamma_A = \gamma_A' \) for each \( A \) since \( \tau_A \) is monic. This implies \( \gamma = \gamma' \)

\( \Rightarrow \): Fix \( A \in |A| \). If \( x \) and \( x' \) from \( F(A) \) satisfy \( \tau_A(x) = \tau_A(x') \) then, with \( y_{A,F}^{-1} \) from 2.49, the following hold for any \( B \in |A| \) and any \( f \in \mathcal{A}(A, B) \):

\[
\tau_B \left( y_{A,F}^{-1}(x)B(f) \right) = \tau_B \left( y_{A,F}^{-1}(x)B(f \circ id_A) \right) = \tau_B \left( y_{A,F}^{-1}(x)B(\tilde{A}(f)(id_A)) \right) = \tau_B \left( F(f)(\tau_{A,F}^{-1}(x)(id_A)) \right) = G(f)(\tau_A(F(id_A)(x))) = G(f)(\tau_A(x)) = G(f)(\tau_A(x')) = \tau_B(y_{A,F}^{-1}(x')B(f))
\]

where naturality of \( y_{A,F}^{-1}(x) \) gives the third equality, naturality of \( \tau \) gives the next, and the last step is obtained by reversing the previous ones. Since \( B \) and \( f \) were arbitrary, this implies

\[
\tau \circ y_{A,F}^{-1}(x) = \tau \circ y_{A,F}^{-1}(x')
\]

which implies \( y_{A,F}^{-1}(x) = y_{A,F}^{-1}(x') \) since \( \tau \) is monic. This implies \( x = x' \) since \( y_{A,F}^{-1} \) is a bijection.

\( \blacksquare \)
2.4. NATURAL TRANSFORMATIONS

2.495 (Yoneda embedding) A a small category implies existence of a functor

\[ Y_A : \mathcal{A} \to \text{Set}^{\mathcal{A}} \]

via \( Y_A(A) = \hat{A} \) and \( Y_A(a) = \hat{a} \). Moreover,

i) \( Y_A \) is full and conservative.

ii) \( Y_A(a) \) is monic iff \( a \) is monic.

iii) if \( Y_A(a) \) is epic so is \( a \). If \( a \) is split epic, \( Y_A(a) \) is epic.

Because: 2.343 implies \( Y_A \) is well-defined on objects. 2.453 implies it is well-defined on morphisms.

\( Y_A \) preserves identities: \( A \in |\mathcal{A}| \) implies \( \hat{id_A} : \hat{A} \Rightarrow \hat{A} : \mathcal{A}^0 \to \text{Set} \) has components \( (\hat{id_A})_{B^o} : \mathcal{A}(B, A) \to \mathcal{A}(B, A) \) defined by \( f \mapsto id_A \circ f = f \).

\( Y_A \) preserves composites: for \( A \xrightarrow{a} A_1 \xrightarrow{a_1} A_2, B^o \in |\mathcal{A}^0|, \) and \( f \in \mathcal{A}(B, A) \),

\[ Y_A(a_1 \circ a)(f) = (a_1 \circ a) \circ f = a_1 \circ (a \circ f) = Y_A(a_1)(a \circ f) = Y_A(a_1)(Y_A(a)(f)). \]

\( Y_A \) is full and faithful: for \( A, A_1 \in |\mathcal{A}| \),

\[ \text{Set}^{\mathcal{A}^0}(Y_A(A), Y_A(A_1)) = \text{Set}^{\mathcal{A}^0}(\hat{A}, \hat{A_1}) \cong \mathcal{A}(A, A_1) \]

where 2.493 gives the second step.

\( Y_A \) reflects isomorphisms: If \( a \in \mathcal{A}(A, A_1) \) is such that \( Y_A(a) = \hat{a} : \mathcal{A}^0 \to \text{Set} \) is a natural isomorphism then there is a natural transformation \( \varphi : \hat{A} \to \hat{A} \) for which

\[ \hat{A} \xrightarrow{\hat{a}} \hat{A_1} \xleftarrow{\varphi} \hat{A} \quad \text{and} \quad \hat{A_1} \xrightarrow{\hat{a}} \hat{A} \xleftarrow{id_{\hat{A}}} \hat{A_1} \]

But \( \varphi = \hat{u} \) for some \( u \in \mathcal{A}(A_1, A) \) since \( Y_A \) is full. \( B^o \in |\mathcal{B}^o| \) implies \( (\hat{u} \circ \hat{a})_{B^o} = id_{\hat{A}(B^o)} = id_{\hat{A}(B)} \).

Because it is faithful, \( Y_A \) reflects monics and epics.

\( Y_A \) preserves monics: given \( a \in \mathcal{A}(A, A_1) \) monic and \( \varphi, \psi : F \Rightarrow \hat{A} : \mathcal{A}^0 \to \text{Set} \) for which \( \hat{a} \circ \varphi = \hat{a} \circ \psi \), fix \( B^o \in |\mathcal{A}^0| \) and fix \( x \in F(B^o) \). \( \hat{a} \circ \varphi = \hat{a} \circ \psi \) implies \( \hat{a}_{B^o} \circ \varphi_{B^o} = \hat{a}_{B^o} \circ \psi_{B^o} \) implies \( \hat{a}_{B^o}(\varphi_{B^o}(x)) = \hat{a}_{B^o}(\psi_{B^o}(x)) \) implies \( a \circ (\varphi_{B^o}(x)) = a \circ (\psi_{B^o}(x)) \) but a monic gives \( \varphi_{B^o}(x) = \psi_{B^o}(x) \). \( x \) and \( B^o \) arbitrary give \( \varphi = \psi \), hence, \( \hat{a} \) monic in \( \text{Set}^{\mathcal{A}^0} \).

\( a \in \mathcal{A}(A, A_1) \) split epic implies

\[ A \xrightarrow{m} A_1 \xrightarrow{id_{\hat{A_1}}} \hat{A_1} \]

which implies \( \hat{a} \) epic in \( \text{Set}^{\mathcal{A}^0} \).

Note that \( Y_{\mathcal{A}^0} : \mathcal{A}^0 \to \text{Set}^{\mathcal{A}} \).
The bijections 2.49 and 2.492 respectively give natural isomorphisms from the upper to lower paths of the following diagrams.

\[ \begin{array}{ccc}
A \times \text{Set}^A & \xrightarrow{(Y_A) \circ \times 1} & \text{Set}^A \circ \times \text{Set}^A \\
\uparrow & & \downarrow \\
\text{Set}^A \times A & \xrightarrow{ev} & \text{Set}
\end{array} \quad \text{and} \quad \begin{array}{ccc}
A^\circ \times \text{Set}^{A^\circ} & \xrightarrow{(Y_A) \circ \times 1} & \text{Set}^{A^\circ} \circ \times \text{Set}^{A^\circ} \\
\uparrow & & \downarrow \\
\text{Set}^{A^\circ} \times A^\circ & \xrightarrow{ev} & \text{Set}
\end{array} \]

## 2.5 Limits

A cone on a functor \( F : \mathcal{B} \rightarrow \mathcal{C} \) is a pair \((L, \lambda)\) with \( L \in |\mathcal{C}| \) and \( \lambda : K_L \Rightarrow F : \mathcal{B} \rightarrow \mathcal{C} \) a natural transformation. That is, \( b \in B(B, B') \) implies

\[ \begin{array}{ccc}
F(B) & \xrightarrow{\lambda_B} & L \\
\downarrow & & \downarrow \\
F(b) & \xrightarrow{\lambda_{B'}} & F(B').
\end{array} \]

### 2.51 Cone Morphisms

Let \((L, \lambda), (L', \lambda')\) be cones on a functor \( F : \mathcal{B} \rightarrow \mathcal{C} \). A cone morphism \( \varphi : (L', \lambda') \rightarrow (L, \lambda) \) is a morphism \( \varphi \in \mathcal{C}(L', L) \) such that \( \forall B \in |\mathcal{B}|, \lambda_B \circ \varphi = \lambda_B' \). That is, \( B \in |\mathcal{B}| \) implies

\[ \begin{array}{ccc}
F(B) & \xrightarrow{\lambda_B} & L \\
\downarrow & & \downarrow \\
F(B') & \xrightarrow{\lambda_{B'}} & L'.
\end{array} \]

It is a cone isomorphism if \( \varphi \in \mathcal{C}(L', L) \) is an isomorphism.

### 2.511 A limit of a functor \( F : \mathcal{B} \rightarrow \mathcal{C} \) is a cone \((L, \lambda)\) on \( F \) such that for each cone \((L', \lambda')\) on \( F \) there is a unique cone morphism \( \varphi : (L', \lambda') \rightarrow (L, \lambda) \).

### 2.512 Let \( F : \mathcal{B} \rightarrow \mathcal{C} \) be a functor. If \((L, \lambda)\) and \((L', \lambda')\) are limits of \( F \) then there is a unique cone isomorphism \( \varphi : (L', \lambda') \rightarrow (L, \lambda) \).

*Because: see 2.6.3 in Volume 1 of [25].*

### 2.513 If \((L, \lambda)\) is a limit of a functor \( F : \mathcal{B} \rightarrow \mathcal{C} \) then \( \{\lambda_B \mid B \in |\mathcal{B}|\} \) is monic in \( \mathcal{C} \).

*Because: 2.21 and 2.6.4 in Volume 1 of [25].*

### 2.52 Terminal Objects

An object \( 1 \) of a category \( \mathcal{C} \) is a terminator (or terminal object) if for any \( \mathcal{C} \)-object \( X \) there is a unique morphism \( !_X : X \rightarrow 1 \). See 1.421 on page 39 of [69].
2.521 For a category $\mathcal{C}$ with terminator $1$, morphisms $1 \to X$ are points of $X$. See 1.4(11)4 of [69].

2.522 A terminator (together with an empty class of projection morphisms) in a category $\mathcal{C}$ may be construed as a limit of the empty functor $\phi \to \mathcal{C}$.

Because: a cone on $\phi \to \mathcal{C}$ is an object $L$. A limit is a cone $1$ such that for any cone $L$ there is a unique morphism $L \to 1$ satisfying no extra conditions.

2.523 Let $1$ be a terminator in a category $\mathcal{C}$. $1'$ is a terminator iff there is a unique isomorphism $1 \to 1'$.

Because: $\Rightarrow$: $1 \xrightarrow{u} 1'$ exists since $1'$ is terminal. Similarly there is $1' \xrightarrow{v} 1$. But $1' \xrightarrow{v} 1 \xrightarrow{u} 1'$ and $1 \xrightarrow{u} 1' \xrightarrow{v} 1$ are identities by definition of terminator. Uniqueness of these isomorphisms also follows from the definition of terminator.

$\Leftarrow$: Let $1 \xrightarrow{u} 1'$ be an isomorphism. If $C \xrightarrow{k} 1'$ and $C \xrightarrow{k'} 1'$ then $u^{-1} \circ k = u^{-1} \circ k'$ implies $k = k'$.

Since a coreflection is a terminator in a particular comma category, coreflections in general and limits in particular are unique to isomorphism.

2.53 Products. A product of a pair $(A, B)$ of objects in a category $\mathcal{C}$ is a triple $(A \times B, \pi_A, \pi_B)$ with $A \times B \in |\mathcal{C}|$ and

$A \xleftarrow{\pi_A} A \times B \xrightarrow{\pi_B} C$

with the property that: for any pair of morphisms $A \xleftarrow{Z} B$, there is a unique $Z \to A \times B$ for which

$\xymatrix{ A \ar[r]_{\pi_A} & A \times B \ar[r]_{\pi_B} & C \ar[dl]^{Z} }$

See 1.423 on page 40 of [69] for a concise, pictorial definition.

2.531 Products may be construed as limits having discrete categories as domains.

Because: see, for example, [102].

2.532 Let $(X, \pi)$ be a product of $\{X_k \mid k \in I\}$ for some index set $I$ such that the projection $X \xrightarrow{\pi_i} X_i$ factors through a terminator:

$\xymatrix{ X \ar[r]^{\pi_i} & X_i \ar@{-->}[dl]^{!} \ar@{-->}[dr]^{p} \ar@{-->}[d]^{1} }$

If $(Y, \gamma)$ is a product of $\{X_k \mid k \in I; k \neq i\}$, then $(Y, \gamma)$ is a product of $\{X_k \mid k \in I\}$ as well.
Because: there exist unique maps $u$ and $v$ for which
\[
\begin{array}{c}
\begin{array}{c}
X_i \xrightarrow{p} X \\
\downarrow \pi_i \\
Y \xrightarrow{u} X \\
\downarrow \gamma_j \\
X_j \\
\end{array}
\end{array}
\begin{array}{c}
\begin{array}{c}
X_j \xrightarrow{\pi_j} Y \\
\downarrow \gamma_j \\
X \xrightarrow{v} Y \\
\end{array}
\end{array}
\]
for each $j \neq i$. Then for each $j \neq i$,
\[
\gamma_j \circ v \circ u = \pi_j \circ u = \gamma_j = \gamma_j \circ id_Y
\]
which implies $v \circ u = id_Y$. Moreover,
\[
\pi_i \circ u \circ v = p \circ ! \circ u \circ v = p \circ ! = \pi_i = \pi_i \circ id_X
\]
and for $j \neq i$,
\[
\pi_j \circ u \circ v = \gamma_j \circ v = \pi_j = \pi_j \circ id_X
\]
which together imply $u \circ v = id_X$.

2.54 Equalizers. A nonempty set of morphisms $E = \{ A \xrightarrow{f_i} B \mid i \in I \}$ is an equation. A solution to an equation is a morphism $E \xrightarrow{e} A$ such that $f_i \circ e = f_j \circ e$ for all $i, j \in I$.

\[
E \xrightarrow{e} A \xrightarrow{f_i} B \\
\]

2.541 A solution $E \xrightarrow{e} A$ to an equation $E = \{ A \xrightarrow{f_i} B \mid i \in I \}$ is an equalizer if for any solution $E' \xrightarrow{e'} A$ of $E$, there is a unique $E' \xrightarrow{\varphi} E$ such that $e \circ \varphi = e'$.

\[
E' \xrightarrow{\varphi} E \xrightarrow{e} A \xrightarrow{f_i} B \\
\]

In particular, an equalizer of a pair $A \xrightarrow{f} B$ of morphisms is a morphism $E \xrightarrow{e} A$ for which $f \circ e = g \circ e$ and such that if $f \circ e' = g \circ e'$, then there is a unique $\varphi$ satisfying $e \circ \varphi = e'$. That is

\[
E' \xrightarrow{e'} A \xrightarrow{f} B \\
E \xrightarrow{e} A \xrightarrow{f} B
\]
For any morphism \( A \xrightarrow{f} B \), \( \{ A \xrightarrow{id_A} A \} \) is an equalizer of \( \{ A \xrightarrow{f} B \} \).

Because: \( e \) gives the unique vertical arrow in

\[
\begin{array}{ccc}
E & \xleftarrow{e} & A \\
\downarrow & & \xrightarrow{id_A} \\
\downarrow & & \xrightarrow{f} \\
A & \xrightarrow{f} & B
\end{array}
\]

An equalizer may be construed as a limit. Because: it is a limit of the functor \( E \xrightarrow{F} C \) with

\[
E = \begin{array}{ccc}
& i \\
\bullet & \circ & j \\
& \circ
\end{array}
\]

and defined by \( F(\bullet) = A \), \( F(\circ) = B \), and \( F(i) = f_i \).

Equalizers are unique up to isomorphism.

Because: this follows from 2.543 and 2.512.

If \( E \xrightarrow{e} A \) is an equalizer then \( e \) is monic.

Because: \( e \circ u = e \circ v \) implies \( f_i \circ (e \circ u) = (f_i \circ e) \circ u = (f_j \circ e) \circ u = f_j \circ (e \circ u) \) for all \( i, j \in I \), hence, there is a unique \( \varphi \) for which \( e \circ \varphi = e \circ u \). Then \( e \circ v = e \circ u \) implies \( u = \varphi = u \).

The converse of 2.545 is false: there are categories in which not every monic is an equalizer. A monic is **regular** if it is an equalizer of some nonempty set of morphisms.

If every monic in \( C \) is regular, then \( C \) is balanced.

Because: if \( E \xrightarrow{e} A \) is monic and epic, then \( e \) is an equalizer \( E \xrightarrow{e} A \) of a nonempty set \( \{ A \xrightarrow{f_i} B \mid i \in I \} \) of morphisms. However, \( f_i \circ e = f_j \circ e \) implies \( f_i = f_j \) since \( e \) is epic. That is, there is a single morphism \( f \) for which \( E \xrightarrow{e} A \xrightarrow{f} B \) is an equalizer. 2.542 and 2.544 then imply that \( e \) is an isomorphism.
2.55 **Pullbacks.** A nonempty set of morphisms \( \mathcal{D} = \left\{ A_i \xrightarrow{f_i} B \mid i \in I \right\} \) with common codomain is called data. A solution to the data is a set \( \mathcal{P} = \left\{ A \xrightarrow{g_i} A_i \mid i \in I \right\} \) such that \( f_i \circ g_i = f_j \circ g_j \) for all \( i, j \in I \).

\[ \begin{array}{c}
  A_i \\
  \downarrow g_i \\
  A \\
  \downarrow g_j \\
  A_j \\
  \downarrow f_j \\
  \end{array} \quad \begin{array}{c}
  B \\
  \downarrow f_i \\
  \end{array} \]

2.551 A solution \( \mathcal{P} = \left\{ A \xrightarrow{g_i} A_i \mid i \in I \right\} \) to data \( \mathcal{D} = \left\{ A_i \xrightarrow{f_i} B \mid i \in I \right\} \) is a pullback if for any solution \( \mathcal{Q} = \left\{ C \xrightarrow{h_i} A_i \mid i \in I \right\} \) there is a unique \( C \xrightarrow{\varphi} A \) such that \( g_i \circ \varphi = h_i \) for all \( i \in I \).

\[ \begin{array}{c}
  C \\
  \downarrow h_i \\
  A_i \\
  \downarrow g_i \\
  A \\
  \downarrow g_j \\
  A_j \\
  \downarrow f_j \\
  \end{array} \quad \begin{array}{c}
  B \\
  \downarrow f_i \\
  \end{array} \]

In particular, a pullback of a pair \( (f, f') \) of morphisms is a pair \( (g, g') \) for which \( f \circ g = f' \circ g' \) and which is such that if \( (r, r') \) satisfies \( f \circ r = f \circ r' \), then there exists a unique \( \varphi \) for which \( g \circ \varphi = r \) and \( g' \circ \varphi = r' \):

\[ \begin{array}{c}
  P \\
  \downarrow g \\
  A \\
  \downarrow f' \\
  C \\
  \downarrow g' \\
  B \\
  \downarrow f \\
  \end{array} \quad \begin{array}{c}
  Q \\
  \downarrow r \\
  \end{array} \quad \begin{array}{c}
  \varphi \\
  \end{array} \]

2.552 For any morphism \( A \xrightarrow{f} B \), \( \left\{ A \xrightarrow{id_A} A \right\} \) is a pullback of \( \left\{ A \xrightarrow{f} B \right\} \).

Because: \( h \) gives the unique vertical arrow in

\[ \begin{array}{c}
  C \\
  \downarrow h \\
  A \\
  \downarrow id_A \\
  A \\
  \downarrow f \\
  B. \end{array} \]
2.553 A pullback may be construed as a limit.

Because: a pullback of \( \{ A_i \xrightarrow{f_i} B \mid i \in I \} \) in a category \( \mathcal{C} \) is a limit of the functor \( \mathcal{I} \xrightarrow{F} \mathcal{C} \) with

![Diagram](image)

and defined by \( F(i) = f_i \).

2.554 Pullbacks are unique up to isomorphism.

Because: this follows from 2.553 and 2.512.

2.555 If \( \{ A \xrightarrow{g_i} A_i \mid i \in I \} \) is a pullback of \( \{ A_i \xrightarrow{f_i} B \mid i \in I \} \), \( f_i \) is monic, and there exists \( j \in I \) with \( j \neq i \), then \( \{ A \xrightarrow{g_j} A_j \mid j \in I, j \neq i \} \) is a monic set of morphisms.

Because: if \( C \xrightarrow{u} A \) and \( g_j \circ u = g_j \circ v \) for all \( j \neq i \), then \( C \xrightarrow{g_k \circ u} A_k \) for all \( k \in I \). Moreover,

\[
    f_k \circ (g_k \circ u) = (f_k \circ g_k) \circ u = (f_{\ell} \circ g_{\ell}) \circ u = f_{\ell} \circ (g_{\ell} \circ u)
\]

for all \( k, \ell \in I \). By definition of pullback, there exists a unique \( C \xrightarrow{\varphi} A \) such that \( g_k \circ \varphi = g_k \circ u \) for all \( k \in I \). By hypothesis, \( g_j \circ v = g_j \circ u \) for \( j \neq i \). Since there exists \( j \in I \) with \( j \neq i \),

\[
    f_i \circ (g_i \circ u) = (f_i \circ g_i) \circ u = (f_j \circ g_j) \circ u = f_j \circ (g_j \circ u) = f_j \circ (g_j \circ v) = (f_j \circ g_j) \circ v = f_i \circ (g_i \circ v).
\]

\( g_i = circu = g_i \circ v \) since \( f_i \) is monic, hence, \( g_k \circ u = g_k \circ v \) for all \( k \in I \). This implies \( v = \varphi = u \).

In particular, if \( A \xrightarrow{g} X \) is a pullback and \( f \) is monic then \( g' \) is monic.

2.556 \( m \in \mathcal{C}(A, B) \) is monic iff \( A \xrightarrow{id} A \) is a pullback.
Because: this follows directly from the definitions of monic and pullback.

This result is particularly useful in the study of dynamic systems: if a representation between categories of dynamic systems preserves pullbacks (in particular, if it has a left adjoint) then it preserves invariant subobjects.

2.557 A pullback of a pair \((f, f)\) is called a kernel pair of \(f\).

2.56 Complete Categories. Let \(J\) be a small category. A category \(C\) is \(J\)-complete if it admits a limit for each functor \(J \to C\). \(C\) is complete if it is \(J\)-complete for every small category \(J\).

2.561 If \(J\) has a coterminator then every category is \(J\)-complete.
Because: 2.11.5 in Volume 1 of [25].

2.562 If \(C\) is a category which admits all limits then \(C\) is a pre-ordered class.
Because: 2.7.1 in Volume 1 of [25].

2.563 If \(C\) is a small category which admits all small limits then \(C\) is a pre-ordered set.
Because: the proof is similar to that of 2.562.

2.564 For a functor \(G : J \to C\) with \(J\) small, let \(|J|\) and \(\text{mor}(J)\) respectively denote the small discrete categories of objects and morphisms of \(J\). Define \(G_0 : |J| \to C\) by \(J \mapsto G(J)\) and \(G_1 : \text{mor}(J) \to C\) by \(j \mapsto G(\text{cod}(j))\). Let \((P, p)\) and \((P', p')\) respectively be a limits of \(G_0\) and \(G_1\). With \(\alpha\) and \(\beta\) unique morphisms for which \(j \in \text{mor}(J)\) implies

\[
\begin{array}{ccc}
G(\text{cod}(j)) & \xrightarrow{p_{\text{cod}(j)}} & P' \\
\downarrow{\alpha} & & \downarrow{\beta} \\
P & + & P' \\
\downarrow{p_{\text{dom}(j)}} & & \downarrow{p'_{\text{dom}(j)}} \\
G(\text{dom}(j)) & \xleftarrow{G(j)} & G(\text{cod}(j))
\end{array}
\]

and with

\[
\begin{array}{ccc}
L & \xrightarrow{e} & P \\
\downarrow{\alpha} & & \downarrow{\beta} \\
P & + & P' \\
\downarrow{p_{\text{dom}(j)}} & & \downarrow{p'_{\text{dom}(j)}} \\
\end{array}
\]

an equalizer,

\[
\begin{array}{ccc}
L & \xrightarrow{p_{\text{dom}(j)}} & G(J)
\end{array}
\]

is a limit of \(G\).
Because: \((L, \{ p_J \circ e \})\) is a cone on \(G\): \(j \in \mathcal{J}(J, J')\) implies
\[
G(j) \circ p_J \circ e = p'_j \circ \beta \circ e = p'_j \circ \alpha \circ e = p_{J'} \circ e
\]
in which definitions of \(\beta\), equalizer, and \(\alpha\) justify the equalities. Given a cone \((C, \gamma)\) on \(G\), the limit \((P, p)\) supplies a unique \(\varphi \in \mathcal{C}(C, P)\) for which \(J \in \mathcal{J}\) implies
\[
\begin{array}{c}
\gamma_J \\
\downarrow p_J \\
C \varphi \\
\downarrow \end{array}
\]
But \(j \in \mathcal{J}(J, J')\) implies
\[
p'_j \circ \alpha \circ \varphi = p_{J'} \circ \varphi = \gamma_{J'} = G(j) \circ \gamma_J = G(j) \circ p_J \circ \varphi = p'_j \circ \beta \circ \varphi
\]
with definitions of \(\alpha\), \(\varphi\), \(\gamma\), \(\varphi\), and \(\beta\) justifying the equalities. This with 2.513 implies \(\alpha \circ \varphi = \beta \circ \varphi\). Hence, there is a unique \(\psi \in \mathcal{C}(C, L)\) for which \(e \circ \psi = \varphi\). \(J \in \mathcal{J}\) implies
\[
p_J \circ e \circ \psi = p_J \circ \varphi = \gamma_J.
\]
If \(J \in \mathcal{J}\) implies \(p_J \circ e \circ \psi' = \gamma_J\) then \(p_J \circ e \circ \psi' = p_J \circ \varphi\) and together these give \(e \circ \psi' = \varphi\), hence, \(\psi = \psi'\) by definition of equalizer.

2.565 \hspace{1em} A category is complete iff it admits all equalizers and small products.

Because: apply 2.564 or see 1.825 on page 141 of [69].

2.57 \hspace{1em} **Functors Preserve Cones.** Let \(I : \mathcal{J} \to \mathcal{B}\) be a functor and let \((L, \lambda)\) be a cone on \(I\). For any functor \(F : \mathcal{B} \to \mathcal{C}\), \((F(L), 1_F \circ \lambda)\) is a cone on \(F \circ I : \mathcal{J} \to \mathcal{C}\).

That is, functors preserve cones. For each \(j \in \mathcal{J}(J, J')\), the lower left diagram is commutative in \(\mathcal{B}\) while the lower right diagram is commutative \(\mathcal{C}\):

\[
\begin{array}{c}
\mathcal{B} : \\
\begin{array}{c}
L \\
\downarrow \lambda_J \\
I(J) \\
\downarrow \lambda_{J'} \\
I(J')
\end{array}
\end{array}
\hspace{2em}
\begin{array}{c}
\mathcal{C} : \\
\begin{array}{c}
F(L) \\
\downarrow F(\lambda_J) \\
F \circ I(J) \\
\downarrow F(\lambda_{J'}) \\
F \circ I(J')
\end{array}
\end{array}
\]

2.571 \hspace{1em} If
\[
\begin{array}{c}
\mathcal{J} \overset{G}{\longrightarrow} \mathcal{A} \overset{X}{\longrightarrow} \mathcal{C} \\
\begin{array}{c}
K \\
\downarrow H
\end{array}
\end{array}
\]
is a commutative diagram of categories and functors (in particular, \( \varphi = \psi^{-1} \)) and \((L, \lambda)\) is a limit of \(X \circ H\) then \((L, \lambda * id_\varphi)\) is a limit of \(X \circ G\).

Because: \((L, \lambda)\) a limit of \(X \circ H\) implies \(\lambda : \Delta(L) \Rightarrow X \circ H : K \to C\) a natural transformation. This implies \(\lambda * id_\varphi : \Delta(L) \circ \varphi \Rightarrow X \circ H \circ \varphi : J \to C\) with implies \(\lambda * id_\varphi : \Delta(L) \Rightarrow X \circ G : J \to C\). Similarly, \((L', \lambda')\) a cone on \(X \circ G\) induces a cone \((L', \lambda' * id_\psi)\) on \(X \circ H\). Uniqueness of the induced cone morphisms \((L', \lambda') \to (L, \lambda * id_\varphi)\) follows from that of the induced \((L', \lambda' * id_\psi) \to (L, \lambda)\).

2.6 Colimits

A cocone on a functor \(F : \mathcal{B} \to \mathcal{C}\) is a pair \((L, \lambda)\) where \(L \in |\mathcal{C}|\) and \(\lambda : F \Rightarrow K_L : \mathcal{B} \to \mathcal{C}\) is a natural transformation.

That is, \(b \in \mathcal{B}(B, B')\) implies

\[
\begin{array}{ccc}
L & \xrightarrow{\lambda_B} & F(B) \\
\downarrow & & \downarrow \\
F(b) & \xleftarrow{\lambda_{B'}} & F(B')
\end{array}
\]

2.61 Functors Preserve Cocones. Let \(F : \mathcal{B} \to \mathcal{C}\) be a functor. \((L, \lambda)\) is a cocone on \(F\) if \((L^\circ, \lambda^\circ)\) is a cone on \(F^\circ : \mathcal{B}^\circ \to \mathcal{C}^\circ\). Similarly, \((L, \lambda)\) is a cone on \(F\) if \((L^\circ, \lambda^\circ)\) is a cocone on \(F^\circ : \mathcal{B}^\circ \to \mathcal{C}^\circ\).

Because: \(\Rightarrow\): \(F^\circ : \mathcal{B}^\circ \to \mathcal{C}^\circ\) is a well-defined functor 2.372 and \(\lambda^\circ : \Delta(L) \Rightarrow F^\circ \Rightarrow F^\circ : \mathcal{B}^\circ \to \mathcal{C}^\circ\).

\(b^\circ \in \mathcal{B}^\circ(B_1^\circ, B_2^\circ)\) implies \(b \in \mathcal{B}(B_1, B_2)\) implies

\[
\begin{array}{ccc}
F(B_2) & \xrightarrow{\lambda_{B_2}} & L \\
\downarrow & & \downarrow \\
F(b) & \xleftarrow{\lambda_{B_1}} & F(B_1)
\end{array}
\quad\Rightarrow\quad
\begin{array}{ccc}
F^\circ(B_2^\circ) & \xrightarrow{\lambda_{B_2}^\circ} & L^\circ \\
\downarrow & & \downarrow \\
F^\circ(b^\circ) & \xleftarrow{\lambda_{B_1}^\circ} & F^\circ(B_1^\circ)
\end{array}
\]

\(\Leftarrow\): similar.

2.611 Let \((L, \lambda), (L', \lambda')\) be cocones on a functor \(F : \mathcal{B} \to \mathcal{C}\). A cocone morphism \(\varphi : (L, \lambda) \to (L', \lambda')\) is a morphism \(\varphi \in \mathcal{C}(L, L')\) such that \(\forall B \in |\mathcal{B}|, \varphi \circ \lambda_B = \lambda'_B\).

That is, \(B \in |\mathcal{B}|\) implies

\[
\begin{array}{ccc}
L' & \xrightarrow{\varphi} & L \\
\downarrow & & \downarrow \\
F(B) & \xleftarrow{\lambda_B} & \lambda_B'
\end{array}
\]
2.6. COLIMITS

2.6.12 A cocone morphism \( \varphi : (L, \lambda) \to (L', \lambda') \) is a cocone isomorphism if \( \varphi \in C(L, L') \) is an isomorphism.

2.6.13 A colimit of a functor \( F : B \to C \) is a cocone \((L, \lambda)\) on \( F \) such that for each cocone \((L', \lambda')\) on \( F \) there is a unique cocone morphism \( \varphi : (L, \lambda) \to (L', \lambda') \).

2.6.14 Let \( F : B \to C \) be a functor. \((L, \lambda)\) is a colimit of \( F \) iff \((L \circ \lambda, \lambda \circ \lambda)\) is a limit of \( F \circ \lambda : B \circ \lambda \to C \circ \lambda \).

Because: \( \Rightarrow \): \((L, \lambda)\) a colimit of \( F \) implies \((L, \lambda)\) a cocone of \( F \) implies \((L \circ \lambda, \lambda \circ \lambda)\) a cone on \( F \). The unique, induced cocone morphism \( \varphi : (L, \lambda) \to (M, \mu) \) induces a unique \( \varphi \circ \lambda : (M \circ \lambda, \mu \circ \lambda) \to (L \circ \lambda, \lambda \circ \lambda) \).

\( \Leftarrow \): similar

2.6.15 Let \( F : B \to C \) be a functor. If \((L, \lambda)\) and \((L', \lambda')\) are colimits of \( F \) then there is a cocone isomorphism \( \varphi : (L, \lambda) \to (L', \lambda') \).

Because: 2.6.14 and 2.5.12.

2.62 Initial Objects. An object \( 0 \) of a category \( C \) is a coterminator (or initial object) if for any \( C \)-object \( X \) there is a unique morphism \( i_X : 0 \to X \).

2.6.21 A coterminator (together with an empty class of morphisms) is a colimit (of an empty functor).

Because: see \([102]\).

2.6.22 0 is a coterminator in \( C \) iff \( 0^\circ \) is a terminator in \( C^\circ \).

Because: 2.6.21, 2.6.14, 2.5.22.

2.6.23 Let \( 0 \) be a coterminator in a category \( A \). For any functor \( F : A \to B \), \((F(0), \{F(1_A)\})\) is a limit.

Because: \((0, \{1_A\})\) a cone on \( \text{id}_A \) implies \((F(0), \{F(1_A)\})\) a cone on \( F \). \((L, \lambda)\) a cone implies existence of a unique \( \varphi \) for which \( \varphi \quad F(A) \quad \lambda \quad L \)

It also implies that \( \varphi = \lambda_0 \) works.

2.6.24 0 is a coterminator in a category \( C \) iff there is a limit \((0, \lambda)\) of \( \text{id}_C \).

Because: \( \Rightarrow \): \((0, \{1_C\})\) is a cone on \( \text{id}_C \). If \((L, \lambda)\) is a cone then \( 0 \to L \) gives the desired cone morphism.
\( (0, \lambda) \) a cone on \( id_C \) implies \( \lambda_C \downarrow \frac{C}{\lambda_C} \) for all \( C \in |C| \). Moreover, \( \lambda_C \downarrow \frac{C}{\lambda_C} \) for all \( C \in |C| \). (0, \lambda) a limit implies \( \lambda_0 = id_0 \). 0 \( \rightarrow \) \( C \rightarrow \lambda_0 \rightarrow \lambda_0 \) 0 since (0, \lambda) is a cone, hence, \( u = \lambda_C \).

2.625 1 is a terminator in a category \( C \) iff there is a colimit \( (1, \lambda) \) of \( id_C \).

Because: 1 is a terminator in \( C \) iff \( 1^\circ \) is a coterminal in \( C^\circ \) iff there is a limit \( (1^\circ, \lambda^\circ) \) of \( id_{C^\circ} \) iff there is a colimit \( (1, \lambda) \) of \( id_C \).

2.63 Coproducts. Coproducts are dual to products and may be construed as special colimits.

2.64 Coequalizers. A coequalizer of morphisms \( X \xrightarrow{\frac{f}{g}} Y \rightarrow Y' \) is a pair \( (Y', k) \) for which

\[
\begin{align*}
X & \xrightarrow{\frac{f}{g}} Y \xrightarrow{k} Y' \\
Z & \xrightarrow{k} Y'
\end{align*}
\]

and

\[
\begin{align*}
X & \xrightarrow{\frac{f}{g}} Y \xrightarrow{k} Y' \\
Z & \xrightarrow{k} Y'
\end{align*}
\]

where the dotted arrow indicates existence of a unique morphism for which the implied commutativity condition holds.

2.641 A coequalizer may be construed as a colimit of a functor having domain the finite category \( \bullet \rightarrow \bullet \rightarrow \bullet \rightarrow \bullet \).

Because: see [102].

2.65 Pushouts. Pushouts are dual to pullbacks and may be construed as special colimits.

2.651 \( k \in C(A, B) \) is epic iff \( A \xrightarrow{k} B \xrightarrow{id_B} B \) is a pushout.
2.66 Cocomplete Categories. A category $\mathcal{C}$ is $\mathcal{J}$-cocomplete if it admits a colimit for each functor $\mathcal{J} \to \mathcal{C}$. $\mathcal{C}$ is cocomplete if it is $\mathcal{J}$-cocomplete for every small category $\mathcal{J}$.

2.661 For a functor $G : \mathcal{J} \to \mathcal{C}$ with $\mathcal{J}$ small, let $|\mathcal{J}|$ and $\text{mor}(\mathcal{J})$ respectively denote the small discrete categories of objects and morphisms of $\mathcal{J}$. Define $G_0 : |\mathcal{J}| \to \mathcal{C}$ by $j \mapsto G(\text{dom}(j))$. Let $(P, p)$ and $(P', p')$ respectively be a colimits of $G_0$ and $G_1$. With $\alpha$ and $\beta$ unique morphisms for which $j \in \text{mor}(\mathcal{J})$ implies

![Diagram]

and with

$P' \xrightarrow{\alpha} P \xrightarrow{\beta} L$ a coequalizer,

$G(J) \xrightarrow{\text{coeq}_J} L$

is a colimit of $G$.

Because: similar to 2.564.

2.662 A category is cocomplete iff it admits all coequalizers and small coproducts.

2.7 Adjoint

A reflection of an object $X$ along a functor $G : \mathcal{A} \to \mathcal{X}$ is a pair $(F(X), \eta_X)$ with $F(X) \in |\mathcal{A}|$ and $\eta_X \in \mathcal{X}(X, G(F(X)))$ such that for any pair $(A, x)$ with $A \in |\mathcal{A}|$ and $x \in \mathcal{X}(X, G(A))$, there is a unique $a \in \mathcal{A}(F(X), A)$ for which $G(a) \circ \eta_X = x$.

![Diagram]

The ingredients of set theory include one type of things, sets, and the binary membership predicate. In category theory, one has morphisms and the ternary predicate: composite. Theorems which keep careful accounts of morphisms are significant, therefore. Mac Lane (page 55 of [102]) refers to $(F(X), \eta_X)$ as a universal arrow from $X$ to $G$. 
2.71  Adjoint as Terminal Objects. \((F(X), \eta_X)\) is a reflection of \(X \in \mathcal{X}\) along \(G : A \to \mathcal{X}\) iff it is a coterminalator in \(X \downarrow G\).

Because: see \([102]\).

2.711  Let \((F(X), \eta_X)\) be a reflection of \(X \in \mathcal{X}\) along a functor \(G : A \to \mathcal{X}\). \((F'(X), \eta'_{X})\) is a reflection of \(X\) along \(G\) iff there is a unique isomorphism \(a \in A(F(X), F'(X))\) for which \(G(a) \circ \eta_X = \eta'_{X}\).

\[
\begin{array}{ccc}
F'(X) & \xleftarrow{a} & F(X) \\
\eta_X & \downarrow & \\
G(F(X)) & \xrightarrow{G(a)} & G(F'(X))
\end{array}
\]

Because: in either case \((F(X), \eta_X)\) and \((F'(X), \eta'_{X})\) are both a coterminalator in \(X \downarrow G\). \(X_G : X \downarrow G \to A\) preserves and reflects isomorphisms.

2.712  \(K : A \to \text{Set}\) is representable iff there exists \(R \in |A|\) and a natural isomorphism \(\varphi : \tilde{R} \cong K\). \(\varphi\) is a representation of \(K\).

2.713  A reflection of \(X \in \mathcal{X}\) along \(G : A \to \mathcal{X}\) exists iff \(\tilde{X} \circ G : A \to \text{Set}\) is representable.

Because: \(\Rightarrow\): \((R, \eta_X)\) a reflection gives \(\varphi_A : \tilde{R}(A) \to \tilde{X}(G(A))\) with \(\varphi_A(u) = G(u) \circ \eta_X\). By definition of reflection, \(\varphi_A\) is a bijection. Naturality is commutativity of

\[
\begin{array}{ccc}
A(R, A) & \xrightarrow{\varphi_A} & \mathcal{X}(X, G(A)) \\
\tilde{R}(a) & \downarrow & \\
A(R, A_1) & \xrightarrow{\varphi_{A_1}} & \mathcal{X}(X, G(A_1))
\end{array}
\]

for each \(a \in A(A, A_1)\). If \(u \in A(R, A)\), then

\[
\varphi_{A_1}(\tilde{R}(a)(u)) = \varphi_{A_1}(a \circ u) = G(a \circ u) \circ \eta_X = G(a) \circ G(u) \circ \eta_X = \tilde{X}(G(a))(G(u) \circ \eta_X) = \tilde{X}(G(a))(\varphi_A(u)) = \tilde{X}(G(a)) \circ \varphi_A(u).
\]

\(
\Leftarrow\): \(\tilde{R} \Rightarrow \tilde{X} : G\) a representation implies \((R, \varphi_R(id_R))\) a reflection of \(X\) along \(G\).

2.714  Mac Lane \([102]\) (page 58) defines a universal element of a functor \(K : A \to \text{Set}\) to be a reflection of any one-point set along \(K\).
2.7.15 Let $F : J \to C$ be a functor with $J$ small. $C$ admits a colimit of $F$ iff it admits a reflection of $F$ along $\Delta : C \to C^J$.

Because: an object of $F \downarrow \Delta$ is precisely a cocone on $F$. Morphisms in $F \downarrow \Delta$ are the cocone morphisms.

2.7.16 A reflection of $X \in |\mathcal{X}|$ along $G : A \to \mathcal{X}$ exists iff $A$ admits a limit of $X \downarrow G \xrightarrow{X_G} A$ and this limit is preserved by $G$.

Because: $\Rightarrow$: $(R, \eta)$ a reflection of $X$ along $G$ implies $(R, \eta)$ a coterm inator in $X \downarrow G$. Any functor maps a coterm inator to a limit object 2.624 hence, $A$ admits a limit of $X_G$. Similarly, $G \circ X_G$ maps this coterm inator to a limit.

$\Leftarrow$: If $G$ preserves a limit for $X_G : X \downarrow G \to A$, it preserves all limits for this functor. These limits are the same as those of $X \downarrow G \xrightarrow{id} X \downarrow G \xrightarrow{X_G} A$, hence, $G$ preserves limits of this composite. ?? implies $X_G$ creates limits for $X \downarrow G \xrightarrow{id} X \downarrow G$. Since $A$ admits a limit of $X_G$, $id_{X \downarrow G}$ has a limit. By 2.624 this is a coterm inator in $X \downarrow G$. By 2.71 this is a reflection of $X$ along $G$.

2.72 Reflective Functors. $G : A \to \mathcal{X}$ is reflective if a reflection along $G$ exists for each $X \in |\mathcal{X}|$. This is a property of a functor $G$. Assuming a sufficiently strong choice axiom it is equivalent the the property of having a left adjoint.

2.721 If $G : A \to \mathcal{X}$ and $G' : \mathcal{X} \to C$ are a reflective functors then so is $G' \circ G : A \to C$.

Because: see 3.2.1 in Volume 1 of [25].

2.722 Let $J$ be a small category. A category $C$ is $J$-cocomplete iff $\Delta : C \to C^J$ is reflective.

Because: 2.715.

2.723 $G : A \to \mathcal{X}$ is reflective iff $A$ admits a limit of each $X_G : X \downarrow G \to A$ for each $X \in |\mathcal{X}|$ and $G$ preserves these limits.

Because: 2.716.

2.73 Reflective Mappings of Actions. Let $G : A \to \mathcal{X}$ be a reflective functor. For any small category $J$, $G_J : A^J \to \mathcal{X}^J$ is reflective.

Because: see 3.2.4 in Volume 1 of [25].

2.731 For any (locally small) category $A$, the hom functors $\hat{A} : A \to \text{Set}$ preserve all limits which exist in $A$ (Theorem 1 on page 116 of [102]). According to Freyd’s Adjoint Functor Theorem, $A$ complete implies such functors have left adjoints if, in addition, the Solution Set Condition holds. Hom functors
need not have adjoints if the completeness hypothesis does not hold. Consider $\mathcal{A} = \begin{tikzpicture}[baseline=(current bounding box.center)]\node (A) at (0,0) {0};\node (B) at (1,0) {1};\node (C) at (0,1) {s};\node (D) at (1,1) {t};\draw[->] (A) to (B);\draw[->] (A) to (C);\draw[->] (A) to (D);\end{tikzpicture}$. There is no reflection of 2 along $\mathcal{A} \rightarrow \textbf{Set}$. Consider:

\[
\begin{array}{c}
\begin{tikzpicture}[baseline=(current bounding box.center)]\node (A) at (0,0) {1};\node (B) at (1,0) {1};\node (C) at (0,1) {\{s, t\}};\node (D) at (1,1) {\{s, t\}};\draw[->] (A) to (B);\draw[->] (A) to (C);\draw[->] (A) to (D);\end{tikzpicture}
\end{array}
\]

with $f$ and $g$ the two bijections $2 \rightarrow \{s, t\}$.

2.74 Coreflections. A coreflection of an object $A$ along a functor $F : \mathcal{X} \rightarrow \mathcal{A}$ is a pair $(G(A), \varepsilon_A)$ with $G(A) \in |\mathcal{X}|$ and $\varepsilon_A \in \mathcal{A}(F(G(A)), A)$ such that for any pair $(X, a)$ with $X \in |\mathcal{X}|$ and $a \in \mathcal{A}(F(X), A)$, there is a unique $x \in \mathcal{X}(X, G(A))$ such that $\varepsilon_A \circ F(x) = a$.

\[
\begin{array}{c}
\begin{tikzpicture}[baseline=(current bounding box.center)]\node (A) at (0,0) {A};\node (B) at (0,-1) {G(A)};\node (C) at (1,0) {F(G(A))};\node (D) at (2,0) {F(X)};\draw[->] (A) to (B);\draw[->] (A) to (C);\draw[->] (A) to (D);\end{tikzpicture}
\end{array}
\]

A coreflection of $A$ along $F$ gives a precise way to specify $\mathcal{X}$-morphisms with codomain $G(A)$.

2.741 $(G(A), \varepsilon_A)$ is a coreflection of $A \in |\mathcal{A}|$ along $F : \mathcal{X} \rightarrow \mathcal{A}$ iff $(G(A)^\circ, \varepsilon_A^\circ)$ is a reflection of $A^\circ \in |\mathcal{A}^\circ|$ along $F^\circ : \mathcal{X}^\circ \rightarrow \mathcal{A}^\circ$.

Because: see [102].

2.742 Let $(G(A), \varepsilon_A)$ and $(G'(A), \varepsilon'_A)$ be coreflections of an object $A$ along a functor $F : \mathcal{X} \rightarrow \mathcal{A}$. There is a unique isomorphism $\psi \in \mathcal{X}(G'(A), G(A))$ such that $\varepsilon_A \circ F(\psi) = \varepsilon'_A$.

\[
\begin{array}{c}
\begin{tikzpicture}[baseline=(current bounding box.center)]\node (A) at (0,0) {A};\node (B) at (0,-1) {G(A)};\node (C) at (1,0) {F(G(A))};\node (D) at (2,0) {F(G'(A))};\draw[->] (A) to (B);\draw[->] (A) to (C);\draw[->] (A) to (D);\end{tikzpicture}
\end{array}
\]

Because: see [102].

2.743 A coreflection of $A \in |\mathcal{A}|$ along $F : \mathcal{X} \rightarrow \mathcal{A}$ exists iff $\hat{A} \circ F^\circ : \mathcal{X}^\circ \rightarrow \textbf{Set}$ is representable.

Because: a coreflection of $A \in |\mathcal{A}|$ along $F : \mathcal{X} \rightarrow \mathcal{A}$ exists iff a reflection of $A^\circ \in |\mathcal{A}^\circ|$ along $F^\circ : \mathcal{X}^\circ \rightarrow \mathcal{A}^\circ$ exists iff $\hat{A}^\circ \circ F^\circ$ is representable iff $\hat{A} \circ F^\circ$ is representable since $\delta_A : \hat{A}^\circ \Rightarrow \hat{A}$ is a natural isomorphism 2.452.
Let $F : J \to C$ be a functor with $J$ small. $C$ admits a limit of $F$ iff it admits a coreflection of $F$ along $\Delta : C \to C^J$.

Because: an object of $\Delta \downarrow F$ is precisely a cone on $F$. Morphisms in $\Delta \downarrow F$ are the cone morphisms.

A coreflection of $A \in |A|$ along $F : \mathcal{X} \to A$ exists iff $\mathcal{X}$ admits a colimit of $F \downarrow A \xrightarrow{F^A} \mathcal{X}$ and this limit is preserved by $F$.

Because: $\mathcal{X}$ admits a coreflection of $A$ along $F$ iff $\mathcal{X}^\circ$ admits a reflection of $A^\circ$ along $F^\circ$ iff $\mathcal{X}^\circ$ admits a limit of $(F^A)^\circ : (F \downarrow A)^\circ \to \mathcal{X}^\circ$ preserved by $F^\circ$.

Coreflective Functors. $F : \mathcal{X} \to A$ is coreflective iff a coreflection along $F$ exists for each $A \in |A|$.

$F : \mathcal{X} \to A$ is coreflective iff $F^\circ : \mathcal{X}^\circ \to A^\circ$ is reflective.

Because: 2.614.

If $F : \mathcal{X} \to A$ and $F' : A \to C$ are coreflective then so is $F' \circ F$.

Because: 2.741 and 2.721.

Let $J$ be a small category. A category $C$ is $J$-complete iff $\Delta : C \to C^J$ is coreflective.

Because: 2.744.

$F : \mathcal{X} \to A$ is coreflective iff $\mathcal{X}$ admits a colimit of each $F^A : A \downarrow \to A \xrightarrow{A^\circ} \mathcal{X}$ for each $A \in |A|$ and $F$ preserves these colimits.

Because: 2.745.

Adjunctions. Let $\mathcal{A}$ and $\mathcal{X}$ be categories. An adjunction from $\mathcal{X}$ to $\mathcal{A}$ is a triple $(F, G, \varphi)$ with $G : \mathcal{A} \to \mathcal{X}$ and $F : \mathcal{X} \to \mathcal{A}$ functors and $\varphi$ a natural isomorphism $\mathcal{A}(F(\_), \_ \Rightarrow \mathcal{X}(\_ \Rightarrow G(\_)))$ from the upper to the lower path of:

$$
\xymatrix{
\mathcal{X}^\circ \times \mathcal{X} \ar[rr]^{1 \times G} & & \mathcal{X}^\circ \times \mathcal{X} \ar[r]^{Y_C \times 1} & \text{Set} \\
\mathcal{A}^\circ \times \mathcal{A} \ar[u]^{1 \times Y^A} & & \text{Set}^\circ \times \mathcal{A}^\circ \ar[r]^{ev_{\mathcal{A}^\circ}} \ar[u] & \text{Set} \\
A^\circ \times A \ar[u]_{F^\circ \times 1} & & \text{Set}^\circ \times \mathcal{A}^\circ \ar[r]^{ev_{\mathcal{A}^\circ}} \ar[u] & \text{Set} \\
A \ar[u]_{1 \times 1} & & \\}
$$

where $Y_C : C \to \text{Set}^C$ is the Yoneda embedding. $F \dashv G$, pronounced ‘$F$ is a left adjoint of $G$’ or ‘$G$ is a right adjoint of $F$’, indicates existence of an adjunction $(F, G, \varphi)$. 
2.761 Let \((F, G, \varphi) : \mathcal{X} \to \mathcal{A}\) be an adjunction. The \textbf{unit} is the natural transformation \(\eta : id_\mathcal{X} \Rightarrow G \circ F\) defined by
\[
\eta_X = \varphi_{X,F(X)}(id_{F(X)})
\]
and the \textbf{counit} is the natural transformation \(\varepsilon : F \circ G \Rightarrow id_\mathcal{A}\) defined by
\[
\varepsilon_A = \varphi_{G(A),A}^{-1}(id_G(A)).
\]

2.762 If \((F, G, \varphi) : \mathcal{X} \to \mathcal{A}\) is an adjunction, then its unit and counit are natural transformations.

\textbf{Because:} see [102].

2.763 The unit \(\eta\) and counit \(\varepsilon\) of an adjunction \((F, G, \varphi) : \mathcal{X} \to \mathcal{A}\) satisfy the following.
\[
(id_G * \varepsilon) \circ (\eta * id_G) = id_G
\]
\[
(\varepsilon * id_F) \circ (id_F * \eta) = id_F
\]

\textbf{Because:} naturality of \(\varphi : A(F(\_), \_ \Rightarrow \mathcal{X}(\_), G(\_))\) implies commutativity of
\[
\begin{array}{ccc}
\mathcal{A}(F(X), A) & \xrightarrow{\varphi_{X, A}} & \mathcal{X}(X, G(A)) \\
\mathcal{A}(F(x), a) & \downarrow & \mathcal{X}(x, G(a)) \\
\mathcal{A}(F(Y), B) & \xrightarrow{\varphi_{Y, B}} & \mathcal{X}(Y, G(B))
\end{array}
\]
for each \(x \in \mathcal{X}(Y, X)\) and \(a \in A(A, B)\). That is,
\[
G(a) \circ \varphi_{\mathcal{X}, A}(h) \circ x = \varphi_{\mathcal{X}, B}(a \circ h \circ F(x))
\]
for any \(h \in \mathcal{A}(F(X), A)\). With \(a = \varepsilon_A, x = id_G(A)\), and \(h = id_{F(G(A))}\) this gives
\[
G(\varepsilon_A) \circ \varphi_{G(A), F(G(A))}(id_{F(G(A))}) = \varphi_{G(A), A}(\varepsilon_A)
\]
\[
G(\varepsilon_A) \circ \eta_G(A) = \varphi_{G(A), A} \left( \varphi_{G(A), A}^{-1}(id_G(A)) \right)
\]
\[
G(\varepsilon_A) \circ \eta_G(A) = id_{G(A)}
\]
\[
[(id_G * \varepsilon) \circ (\eta * id_G)]_A = (id_G)_A
\]
\[
(id_G * \varepsilon) \circ (\eta * id_G) = id_G
\]
where 2.761 justifies equality one. Naturality of \(\varphi\) also gives
\[
\varphi_{Y,B}^{-1}(G(a) \circ \varphi_{\mathcal{X}, A}(h) \circ x) = a \circ h \circ F(x).
\]
With \(x = \eta_X, a = id_{F(X)}\), and \(h = \varepsilon_{F(X)}\) this gives
\[
\varphi_{X,F(X)}^{-1} \left( \varphi_{G(F(X), F(X)}(\varepsilon_{F(X)}) \circ \eta_X \right) = \varepsilon_{F(X)} \circ F(\eta_X)
\]
\[
\varphi_{X,F(X)}^{-1}(\eta_X) = \varepsilon_{F(X)} \circ F(\eta_X)
\]
\[
id_{F(X)} = \varepsilon_{F(X)} \circ F(\eta_X)
\]
\[
(id_F)_X = [(\varepsilon * id_F) \circ (id_F * \eta)]_X
\]
\[
id_F = (\varepsilon * id_F) \circ (id_F * \eta)
\]
where the definitions 2.761 of $\varepsilon$ and $\eta$ respectively justify equalities two and three.

\textbf{2.764} If $\mathcal{X} \xrightarrow{F} \mathcal{A}$ are functors and $\eta : id_X \Rightarrow G \circ F$ and $\varepsilon : F \circ G \Rightarrow id_A$ are natural transformations for which

$$(id_G \ast \varepsilon) \circ (\eta \ast id_G) = id_G$$

$$(\varepsilon \ast id_F) \circ (id_F \ast \eta) = id_F$$

then $\eta$ and $\varepsilon$ are respectively the unit and counit of an adjunction $(F, G, \varphi) : \mathcal{X} \to \mathcal{A}$. Moreover,

$$\varphi : \mathcal{A}(F(\underline{\ }), \underline{\ }) \Rightarrow \mathcal{X}(\underline{\ }, G(\underline{\ }))$$

is defined by either

$$\varphi_{X,A}(a) = G(a) \circ \eta_X$$

for $a \in \mathcal{A}(F(X), A)$ or

$$\varphi^{-1}_{X,A}(x) = \eta_A \circ F(x)$$

for $x \in \mathcal{X}(X, G(A))$.

Because: see [102].

\textbf{2.765} Assuming a sufficiently strong axiom of choice, the following conditions on a functor $G : \mathcal{A} \to \mathcal{X}$ are equivalent.

i) $G$ is reflective;

ii) there is an adjunction $(F, G, \varphi)$ from $\mathcal{X}$ to $\mathcal{A}$.

iii) there are a functor $F : \mathcal{X} \to \mathcal{A}$, a natural transformation $\eta : 1_X \Rightarrow G \circ F$ and a natural transformation $\varepsilon : F \circ G \Rightarrow 1_A$ such that

$$(1_G \ast \varepsilon) \circ (\eta \ast 1_G) = 1_G \quad \text{and} \quad (\varepsilon \ast 1_F) \circ (1_F \ast \eta) = 1_F;$$

iv) there are a functor $F : \mathcal{X} \to \mathcal{A}$ and a natural transformation $\eta : 1_X \Rightarrow G \circ F$ such that $X \in |\mathcal{X}|$, implies $(F(X), \eta_X)$ is a reflection of $X$ along $G$;

v) there are a functor $F : \mathcal{X} \to \mathcal{A}$ and a natural transformation $\varepsilon : F \circ G \Rightarrow 1_A$ such that $A \in |\mathcal{A}|$, implies $(G(A), \varepsilon_A)$ is a coreflection of $A$ along $F$.

Because: page 83 of [102].

\textbf{2.766} Let $\mathcal{X}$ and $\mathcal{A}$ be categories. If $(F, G, \eta, \varepsilon)$ is an equivalence then there are adjunctions $(F, G, \varphi) : \mathcal{X} \to \mathcal{A}$ and $(G, F, \psi) : \mathcal{A} \to \mathcal{X}$.

Because: see [102].
2.767 Let \( F : A \to B, F' : B \to C, G' : C \to B \) and \( G : B \to A \) be functors such that \( F \dashv G \) and \( F' \dashv G' \). Then \( F \circ F' \dashv G' \circ G \).

Because: 3.2.1 in Volume 1 of [25].

2.768 A left adjoint is continuous and a right adjoint is cocontinuous.

Because: 1.83 on pages 143–4 of [69].

2.769 If \( \xymatrix{ A \ar[r]^F & X } \) has a left adjoint then it preserves monics. If \( \xymatrix{ X \ar[r]^G & A } \) has a right adjoint then it preserves epics.

Because: 2.768, 2.556, and 2.651.

2.8 Kan extensions

Let \( F : A \to B \) be a functor and let \( C \) be a category. Any functor \( Y : B \to C \) induces \( Y \circ F : A \to C \). If \( A \) and \( B \) are small, this gives the object part of a functor \( C^F : C^B \to C^A \) with \( \tau \mapsto \tau \ast id_F \) on morphisms.

\[
\begin{array}{ccc}
A & \xymatrix{ F \ar[d] & B \\
& C & \end{array}
\]

There are two specifications, called left and right Kan extensions, of functors \( B \to C \) induced by a functor \( A \to C \).

Observe that a coreflection of \( X : A \to C \) along \( C^F \) is a pair \((Y, \epsilon)\) with \( Y : B \to C \) and \( \epsilon : Y \circ F \Rightarrow X \) such that: for every \( \gamma : Z \circ F \Rightarrow X \) with \( Z : B \to C \), there is a unique \( \hat{\gamma} : Z \Rightarrow Y \) for which \( \epsilon \circ (\hat{\gamma} \ast id_F) = \gamma \).

This motivates the following definition which applies even when \( A \) or \( B \) is not small.

2.81 Right Kan Extensions. Let \( F : A \to B \) and \( X : A \to C \) be functors. A right Kan extension of \( X \) along \( F \) is a pair \((Y, \epsilon)\) with \( Y : B \to C \) a functor and \( \epsilon : Y \circ F \Rightarrow X \) a natural transformation such that: for any such pair \((Z, \gamma)\) there is a unique natural transformation \( \hat{\gamma} : Z \Rightarrow Y \) satisfying \( \epsilon \circ (\hat{\gamma} \ast id_F) = \gamma \).

\[
\begin{array}{ccc}
Z & \xymatrix{ \ar[l]^{\hat{\gamma}} & Y \\
& Y \circ F \ar[l]_{\gamma} & \end{array}
\]

2.811 Let \( F : A \to B \) and \( X : A \to C \) be functors. If \((Y, \epsilon)\) and \((Y', \epsilon')\) are right Kan extensions of \( X \) along \( F \) then there is a unique natural isomorphism \( \varphi : Y' \Rightarrow Y : B \to C \) for which \( \epsilon \circ (\varphi \ast id_F) = \epsilon' \). Moreover, \( \epsilon' \circ (\varphi^{-1} \ast id_F) = \epsilon \).
Because: \((Y, \epsilon)\) a right Kan extension implies that there is a unique \(\hat{\epsilon}'\) for which

\[
\begin{array}{c}
Y'' \\
\downarrow \hat{\epsilon}'
\end{array}
\xrightarrow{
\begin{array}{c}
\text{Ran}
\end{array}
}\n\begin{array}{c}
Y
\end{array}
\]

\((Y', \epsilon')\) a right Kan extension implies that there is a unique \(\hat{\epsilon}\) for which

\[
\begin{array}{c}
Y \\
\downarrow \hat{\epsilon}
\end{array}
\xrightarrow{
\begin{array}{c}
\text{Ran}
\end{array}
}\n\begin{array}{c}
Y'
\end{array}
\]

The usual type of argument gives \(\hat{\epsilon} \circ \hat{\epsilon}' = id_{Y'}\) and \(\hat{\epsilon}' \circ \hat{\epsilon} = id_{Y}\).

2.812 Let \(F : A \rightarrow B\) be a functor with \(A\) and \(B\) small. A right Kan extension along \(F\) exists for each \(X : A \rightarrow C\) iff \(C^F : C^B \rightarrow C^A\) has a right adjoint.

Because: 2.765.

2.813 Let \(F : A \rightarrow B\) and \(X : A \rightarrow C\) be functors. A right Kan extension of \(X\) along \(F\) exists if \(B \in |B|\) implies \(C\) admits a limit of \(X \circ B_F : B \downarrow F \rightarrow C\).

Because: the proof proceeds from a sufficiently strong axiom of choice. \((\text{Ran}_F(X), \epsilon)\) will denote the right Kan extension obtained. For each \(B \in |B|\) select a limit \((\text{Ran}_F(X)(B), \lambda^B)\) of \(X \circ B_F : B \downarrow F \rightarrow C\).

For \(b \in B(B, B')\), \((A', \beta') \in |B| \downarrow F\) implies \((A', \beta' \circ b) \in |B| \downarrow F\) and \(X \circ B_F(A', \beta' \circ b) = X(A') = X \circ B_F'(A', \beta')\). \((\text{Ran}_F(X)(B), \{\lambda^B_{(A', \beta', \circ b)} | (A', \beta') \in |B| \downarrow F\})\) is a cone on \(X \circ B_F'\). Let \(\text{Ran}_F(X)(b)\) be the unique, induced map:

\[
\begin{array}{c}
\text{Ran}_F(X)(B) \\
\downarrow \lambda^B_{(A', \beta', \circ b)}
\end{array}
\xrightarrow{
\begin{array}{c}
\text{Ran}_F(X)(b)
\end{array}
}\n\begin{array}{c}
\text{Ran}_F(X)(B')
\end{array}
\]

and

\[
\begin{array}{c}
\text{Ran}_F(X)(B) \\
\downarrow \lambda^B_{(A', \beta', \circ id)}
\end{array}
\xrightarrow{id}
\begin{array}{c}
\text{Ran}_F(X)(B)
\end{array}
\]

\[
\begin{array}{c}
\text{Ran}_F(X)(B') \\
\downarrow \lambda^B_{(A', \beta', \circ \circ b)}
\end{array}
\xrightarrow{\text{Ran}_F(X)(b)}
\begin{array}{c}
\text{Ran}_F(X)(B')
\end{array}
\]

\[
\begin{array}{c}
\text{Ran}_F(X)(B) \\
\downarrow \lambda^B_{(A', \beta', \circ \circ b)}
\end{array}
\xrightarrow{\text{Ran}_F(X)(b)}
\begin{array}{c}
\text{Ran}_F(X)(B')
\end{array}
\]

\[
\begin{array}{c}
\text{Ran}_F(X)(B') \\
\downarrow \lambda^B_{(A', \beta', \circ \circ b)}
\end{array}
\xrightarrow{\text{Ran}_F(X)(b)}
\begin{array}{c}
\text{Ran}_F(X)(B'').
\end{array}
\]
Define $\epsilon : \text{Ran}_F(X) \circ F \Rightarrow X$ by

$$\epsilon_A = \lambda^{F(A)}_{(A, \text{id}_{F(A)})}.$$ 

$\epsilon$ is natural: $a \in \mathcal{A}(A, A')$ implies

$$F(A) \xrightarrow{id_{F(A)}} F(a) \xrightarrow{F(a)} F(A').$$

which implies

$$\text{Ran}_F(X)(F(A)) \xrightarrow{\lambda^{F(A)}_{(A, \text{id}_{F(A)})}} X(A) \xrightarrow{X(a)} X(A').$$

while the definition of $\text{Ran}_F(X)$ gives

$$\text{Ran}_F(X)(F(A)) \xrightarrow{\lambda^{F(A)}_{(A', \text{id}_{F(A')})}} X(A') \xrightarrow{\lambda^{F(A')}_{(A', \text{id}_{F(A')})}} \text{Ran}_F(X)(F(A')).$$

hence,

$$\text{Ran}_F(X)(F(A)) \xrightarrow{\lambda^{F(A)}_{(A, \text{id}_{F(A)})}} X(A) \xrightarrow{X(a)} X(A').$$

Given $Y : B \rightarrow C$ and $\gamma : Y \circ F \Rightarrow X$, $B \in |\mathcal{B}|$ implies $(Y(B), \{\gamma_A \circ Y(b) \mid (A, b) \in |B \downarrow F|\})$ is a cone on $X \circ B_F$. For $a \in \mathcal{A}(A, A')$, naturality of $\gamma$ implies

$$Y(F(A)) \xrightarrow{\gamma_A} X(A)$$

$$Y(F(a)) \xrightarrow{\gamma_{A'}} X(A')$$

hence,

$$F(A) \xrightarrow{F(a)} F(A').$$
implies
\[ X(a) \circ \gamma_A \circ Y(\beta) = \gamma_{A'} \circ Y(F(a)) \circ Y(\beta) = \gamma_{A'} \circ Y(F(a) \circ \beta) = \gamma_{A'} \circ Y(\beta') \]
as depicted in
\[
\begin{array}{c}
X(A) \\
\downarrow \\
Y(B) \\
\downarrow \\
X(A').
\end{array}
\]

Let \( \gamma_B : Y(B) \to \text{Ran}_F(X)(B) \) be the unique, induced morphism:
\[
\begin{array}{c}
\gamma_A \circ Y(\beta) \\
\downarrow \\
X(A) \\
\downarrow \\
\text{Ran}_F(X)(B).
\end{array}
\]

These components give a natural transformation \( \gamma : Y \Rightarrow \text{Ran}_F(X) \) for \( (A', \beta') \in |B'| \downarrow F \), the definition of \( \text{Ran}_F(X) \) justifies the first equality of
\[
\lambda^{B'}_{(A', \beta')} \circ \text{Ran}_F(X)(Y(b)) \circ \gamma_B = \lambda^B_{(A', \beta' \circ b)} \circ \gamma_B
\]
while the definition of \( \gamma \) justifies the second and fourth. These imply \( \text{Ran}_F(X)(b) \circ \gamma_B = \gamma_B \circ Y(b) \) as depicted in
\[
\begin{array}{c}
Y(B) \xrightarrow{\gamma_B} \text{Ran}_F(X)(B) \\
\downarrow Y(b) \\
Y(B') \xrightarrow{\gamma_B'} \text{Ran}_F(X)(B').
\end{array}
\]

Moreover, \( \gamma \) satisfies
\[
[\epsilon \circ (\gamma \ast id_F)]_A = \epsilon_A \circ \gamma_{F(A)} = \lambda^F_{(A, id_F(A))} \circ \gamma_{F(A)} = \gamma_A \circ Y(id_F(A)) = \gamma_A.
\]
This determines \( \gamma \) uniquely: if \( \psi : Y \Rightarrow \text{Ran}_F(X) \) satisfies \( \epsilon \circ (\psi \ast id_F) = \gamma \) then \( A \in |A| \) implies
\[
\lambda^F_{(A, id_F(A))} \circ \psi_{F(A)} = \gamma_A
\]
and \((A, \beta) \in |B \downarrow F|\) implies
\[
\lambda_{(A, \beta)}^B \circ \widehat{\gamma}_B = \gamma_A \circ Y(\beta) = \lambda_{(A, A)}^F \circ \psi_F(A) \circ Y(\beta) = \lambda_{(A, \beta)}^F \circ \text{Ran}_F(X)(\beta) \circ \psi_B = \lambda_{(A, 1)}^B \circ \psi_B
\]
where the definition of \(\widehat{\gamma}\) justifies the first equality, naturality of \(\psi\) justifies the third, and the definition of \(\text{Ran}_F\) justifies the last. Together these imply \(\widehat{\gamma} = \psi\).

In using this to compute adjoints, one must be able to compute certain limits.

2.814 Let \(F : A \to B\) and \(X : A \to C\) be functors with \(A\) and \(B\) small. \(C^F : C^B \to C^A\) has a right adjoint if \(C\) admits certain limits: specifically for each \(B \in |B|\), \(C\) admits a limit of \(B \downarrow F\).

Because: existence of such limits implies existence of a right Kan extension for each \(X : A \to C\) 2.813. 2.765 provides an adjoint.

2.815 Let \((\text{Ran}_F(X), \epsilon)\) be a right Kan extension of \(X : A \to C\) along \(F : A \to B\). If \(F\) is full and faithful then \(\epsilon : \text{Ran}_F(X) \circ F \Rightarrow X : A \to C\) is a natural isomorphism.

Because: \((F(A), id_{F(A)}) \in |F(A) \downarrow F|\). \((A_1, b) \in |F(A) \downarrow F|\) gives
\[
\begin{array}{c}
F(A) \\
frac{id_{F(A)}}{b} \\
F(A) \\
\end{array}
\begin{array}{c}
F(A) \\
\frac{F(a_b)}{F(A_1)}
\end{array}
\]
This object is a coterminal: \(F\) full and faithful implies existence of a unique \(A \xrightarrow{a_b} A_1\) for which
\[
\begin{array}{c}
F(A) \\
\frac{id_{F(A)}}{b} \\
F(A) \\
\end{array}
\begin{array}{c}
F(A) \\
\frac{F(a_b)}{F(A_1)}
\end{array}
\]
This implies \((X(A), \{X(a_b)\}_{(A_1, b)})\) is a limit of \(X \circ F(A)_F\). Since the Right Kan extension is constructed as this limit, there is a unique isomorphism \(\varphi\) for which
\[
\begin{array}{c}
X(A) \\
\frac{X(a_b)}{\text{Ran}_F(X)(F(A))}
\end{array}
\begin{array}{c}
X(A) \\
\frac{\lambda_{(A, b)}^{F(A)}}{\text{Ran}_F(X)(F(A))}
\end{array}
\]
In particular,
since \( a_{id_{F(A)}} = id_A \). This implies \( \epsilon_A = \lambda^{F(A)}_{(A, id_{F(A)})} \) is an isomorphism.

**2.8.16** \( C \) admits a limit of \( X : A \to C \) iff there exists a right Kan extension of \( X \) along \( A \to 1 \).

*Because:* \( \Rightarrow \): A functor \( R : 1 \to C \) is a \( C \)-object and a natural transformation \( R \circ ! \Rightarrow X \) is a cone on \( X \).

*\( \Leftarrow \):* The one comma category of interest is isomorphic to \( A \).

**2.8.17** Given categories and functors

\[
\begin{array}{ccc}
A & \xrightarrow{F} & B \\
\downarrow{X} & & \downarrow{C} \\
C & \xrightarrow{H} & D \\
\end{array}
\]

the functor \( H \) preserves right Kan extensions if \((\text{Ran}_F(X), \epsilon)\) a right Kan extension of \( X \) along \( F \) implies \((H \circ \text{Ran}_F(X), id_H \circ \epsilon)\) a right Kan extension of \( H \circ X \) along \( F \).

**2.8.18** \( G : A \to \mathcal{X} \) has a left adjoint iff there is a right Kan extension of \( id_A \) along \( G \) preserved by \( G \).

\[
\begin{array}{ccc}
A & \xrightarrow{G} & \mathcal{X} \\
\downarrow{id_A} & & \downarrow{\mathcal{X}} \\
\mathcal{A} & \xrightarrow{G} & \mathcal{X} \\
\end{array}
\]

*Because:* \( \Rightarrow \): \((F, G, \varphi)\) an adjunction from \( \mathcal{X} \) to \( A \) has an associated unit \( \eta : id_{\mathcal{X}} \Rightarrow G \circ F \) and counit \( \epsilon : F \circ G \Rightarrow id_A \). \((F, \epsilon)\) gives a right Kan extension. Given

\[
\begin{array}{ccc}
K & \xrightarrow{F} & F \circ G \\
\downarrow{\gamma} & & \downarrow{\gamma \circ G} \\
F \circ G & \xrightarrow{K \circ G} & K \circ G \\
\end{array}
\]

\((\gamma \circ id_F) \circ (id_K \circ \eta)\) has the desired universal property. \( A \in |A| \) gives

\[
[e \circ ([(\gamma \circ id_F) \circ (id_K \circ \eta)] \circ id_G)]_A = e_A \circ [(\gamma \circ id_F)_{G(A)} \circ (id_K \circ \eta)_{G(A)}] = e_A \circ \gamma_{F(G(A))} \circ K(\eta_{G(A)}) = \gamma_A \circ K(G(e_A) \circ \eta_{G(A)}) = \gamma_A
\]
where naturality of $\gamma$ gives equality two and $\text{2.765}$ gives the last equality. To show uniqueness, assume $\sigma : K \Rightarrow F$ satisfies $\varepsilon \circ (\sigma \ast \text{id}_G) = \gamma$. This implies

$$
\begin{align*}
\varepsilon_A \circ \sigma_{G(A)} &= \gamma_A \\
\varepsilon_{F(X)} \circ \sigma_{G(F(X))} &= \gamma_{F(X)} \\
\varepsilon_{F(X)} \circ \sigma_{G(F(X))} \circ K(\eta_X) &= \gamma_{F(X)} \circ K(\eta_X) \\
\sigma_X &= \gamma_{F(X)} \circ K(\eta_X)
\end{align*}
$$

where selecting $A = F(X)$ justifies equality two and naturality of $\sigma$ justifies equality three.

The Kan extension is preserved by $G$: given

\begin{align*}
S & \quad G \circ F \\
\downarrow \mu & \quad \downarrow \mu \\
G \circ F \circ G & \quad S \circ G
\end{align*}

if $\tilde{\mu} = (\mu \ast \text{id}F) \circ (\text{id}_S \ast \eta) : \mathcal{X} \rightarrow \mathcal{X}$ then

$$
\begin{align*}
[[\text{id}_G \ast \varepsilon] \circ (\tilde{\mu} \ast \text{id}_G)]_A = G(\varepsilon_A) & \circ \tilde{\mu}_{G(A)} \\
& = G(\varepsilon_A) \circ \mu_{F(G(A))} \circ S(\eta_{G(A)}) \\
& = \mu_A \circ S(G(\varepsilon_A)) \circ S(\eta_{G(A)}) \\
& = \mu_A
\end{align*}
$$

where naturality of $\mu$ justifies equality three and a property of the unit and counit of an adjunction justifies the last. Uniqueness holds as well since $\mu = (\text{id}_G \ast \varepsilon) \circ (\omega \ast \text{id}_G)$ implies

$$
\begin{align*}
\mu_{F(X)} &= G(\varepsilon_{F(X)}) \circ \omega_{G(F(X))} \\
\mu_{F(X)} \circ S(\eta_X) &= G(\varepsilon_{F(X)}) \circ \omega_{G(F(X))} \circ S(\eta_X) \\
& = G(\varepsilon_{F(X)}) \circ G(F(\eta_X)) \circ \omega_X \\
& = \omega_X
\end{align*}
$$

where naturality of $\omega$ implies equality three while the adjunction property implies the last.

$\Leftarrow$: Let $(F, \varepsilon)$ be a right Kan extension of $\text{id}_A$ along $G$. By hypothesis, $(G \circ F, \text{id}_G \ast \varepsilon)$ is a right Kan extension of $G$ along itself. This implies existence of a unique $\eta : \text{id}_X \rightarrow G \circ F$ for which $(\text{id}_G \ast \varepsilon) \circ (\eta \ast \text{id}_G) = \text{id}_G$:

\begin{align*}
\text{id}_X & \quad G \circ F \\
\downarrow \varepsilon & \quad \downarrow \varepsilon \\
F & \quad F \circ G
\end{align*}

Note that

\begin{align*}
G & \quad \text{id}_G \ast \varepsilon \\
\downarrow \varepsilon & \quad \downarrow \varepsilon \\
F & \quad F \circ \text{id}_F \ast \text{id}_G
\end{align*}
2.8. KAN EXTENSIONS

and that \( \mu = (\varepsilon \ast id_F) \circ (id_F \ast \eta) \) implies

\[
\varepsilon_A \circ \mu_{G(A)} = \varepsilon_A \circ \varepsilon_{F(G(A))} \circ F(\eta_{G(A)}) = \varepsilon_A \circ F(\varepsilon_A) \circ F(\eta_{G(A)}) = \varepsilon_A
\]

where naturality of \( \varepsilon \) implies equality two and a property of adjunctions implies the last. Uniqueness in the definition of Kan extension implies \((\varepsilon \ast id_F) \circ (id_F \ast \eta) = id_F\).

2.82 Left Kan Extensions. Let \( F : A \to B \) and \( X : A \to C \) be functors. A left Kan extension of \( X \) along \( F \) is a pair \((Y, \eta)\) with \( Y : B \to C \) a functor and \( \eta : X \Rightarrow Y \circ F \) a natural transformation such that: for any such pair \((Z, \gamma)\) there is a unique natural transformation \( \tilde{\gamma} : Y \Rightarrow Z \) satisfying \((\tilde{\gamma} \ast id_F) \circ \eta = \gamma \).

2.821 Let \( F : A \to B \) be a functor with \( A \) and \( B \) small. A left Kan extension along \( F \) exists for each \( X : A \to C \) iff \( C^F : C^B \to C^A \) has a left adjoint.

Because: 2.765.

2.822 Let \( F : A \to B \) and \( X : A \to C \) be functors. \((Y, \eta)\) is a left Kan extension of \( X \) along \( F \) iff \((Y^\circ, \eta^\circ)\) is a right Kan extension of \( X^\circ \) along \( F^\circ \).

Because: \( \Rightarrow \) \((Y, \eta)\) a left Kan extension along \( F \) implies \( Y : B \to C \) and \( \eta : X \Rightarrow Y \circ F : A \to C \) implies \( Y^\circ : B^\circ \to C^\circ \) and \( \eta^\circ : (Y \circ F)^\circ \Rightarrow X^\circ : A^\circ \to C^\circ \), that is, \( \eta^\circ : Y^\circ \circ F^\circ \Rightarrow X^\circ : A^\circ \to C^\circ \). If \( Z^\circ : B^\circ \to C^\circ \) and \( \gamma^\circ : Y^\circ \circ F^\circ \Rightarrow X^\circ \), the definition of left Kan extension implies existence of a unique \( \tilde{\gamma} : Y \Rightarrow Z : B \to C \) for which \((\tilde{\gamma} \ast id_F) \circ \eta = \gamma \). This implies \( \tilde{\gamma}^\circ : Z^\circ \Rightarrow Y^\circ \circ F^\circ : B^\circ \to C^\circ \) is such that \((\tilde{\gamma}^\circ \ast id_F^\circ) \circ \eta^\circ = \gamma^\circ \) which implies \( \eta^\circ \circ (\gamma^\circ \ast id_F^\circ) = \gamma^\circ \) 2.443. Uniqueness of \( \tilde{\gamma}^\circ \) follows from that of \( \tilde{\gamma} \).

\( \Leftarrow \) similar.

2.823 Let \( F : A \to B \) and \( X : A \to C \) be functors. If \((Y, \eta)\) and \((Y', \eta')\) are left Kan extensions of \( X \) along \( F \) then there is a unique natural isomorphism \( \varphi : Y \Rightarrow Y' : B \to C \) for which \((\varphi \ast id_F) \circ \eta = \eta' \). Moreover, \( (\varphi^{-1} \ast id_F) \circ \eta' = \eta \).

Because: 2.822 and 2.811.

2.824 Let \( F : A \to B \) and \( X : A \to C \) be functors. A left Kan extension of \( X \) along \( F \) exists if \( B \in |B| \) implies \( C \) admits a colimit of \( X \circ_F B : F \downarrow B \to C \).

Because: a proof may be constructed from 2.822, 2.813, 2.484.
A direct proof proceeds from a sufficiently strong axiom of choice. \((\text{Lan}_F(X), \eta)\) will denote the left Kan extension obtained. For each \(B \in |\mathcal{B}|\) select a colimit \((\text{Lan}_F(X)(B), \lambda^B)\) of \(X \circ F B : F \downarrow B \to \mathcal{C}\).

For \(B \in \mathcal{B}(B, B')\), \((A, \beta) \in |F \downarrow B|\) implies \((A, b \circ \beta) \in |F \downarrow B'|\) and \(X \circ F B(A, \beta) = X(A) = X \circ F B'(A, b \circ \beta)\). \((\text{Lan}_F(X)(B'), \{\lambda^B_{(A, b \circ \beta)} \mid (A, \beta) \in |F \downarrow B|\})\) is a cocone on \(X \circ F B'\). Let \(\text{Lan}_F(X)(b)\) be the unique, induced morphism:

\[
\begin{array}{ccc}
X \circ F B(A, \beta) & \xrightarrow{\lambda^B_{(A, \beta)}} & \text{Lan}_F(X)(B) \\
\downarrow & \downarrow & \downarrow \text{Lan}_F(X)(b) \\
\text{Lan}_F(X)(B') & \xrightarrow{\lambda^B_{(A, b \circ \beta)}} & \text{Lan}_F(X)(B')
\end{array}
\]

\(\text{Lan}_F(X)\) so defined is a functor \(\mathcal{B} \to \mathcal{C}\) since

\[
\begin{array}{ccc}
X \circ F B(A, \beta) & \xrightarrow{\lambda^B_{(A, \beta)}} & \text{Lan}_F(X)(B) \\
\downarrow & \downarrow & \downarrow id \\
\text{Lan}_F(X)(B') & \xrightarrow{\lambda^B_{(A, b \circ \beta)}} & \text{Lan}_F(X)(B)
\end{array}
\]

and

\[
\begin{array}{ccc}
X(A) & \xrightarrow{\lambda^B_{(A, \beta)}} & \text{Lan}_F(X)(B) \\
\downarrow & \downarrow & \downarrow \text{Lan}_F(X)(b) \\
\text{Lan}_F(X)(B') & \xrightarrow{\lambda^B_{(A, b \circ \beta)}} & \text{Lan}_F(X)(B')
\end{array}
\]

\[
\begin{array}{ccc}
X(A) & \xrightarrow{\lambda^B_{(A, b \circ \beta)}} & \text{Lan}_F(X)(B) \\
\downarrow & \downarrow & \downarrow \text{Lan}_F(X)(b \circ \beta) \\
\text{Lan}_F(X)(B') & \xrightarrow{\lambda^B_{(A, b \circ \beta)}} & \text{Lan}_F(X)(B')
\end{array}
\]

Define \(\eta : X \Rightarrow \text{Lan}_F(X) \circ F\) by

\[
\eta_A = \lambda^F(A)_{(A, \text{id}_F(A))}.
\]

\(\eta\) is natural: \(a \in \mathcal{A}(A, A')\) implies

\[
\begin{array}{ccc}
F(A) & \xrightarrow{F(a)} & F(A') \\
\downarrow & \downarrow id & \downarrow \text{id} \\
F(A') & \xrightarrow{id} & F(A')
\end{array}
\]

which implies

\[
\begin{array}{ccc}
X(A) & \xrightarrow{\lambda^F(A')_{(A', \text{id}_F(A'))}} & \text{Lan}_F(X)(F(A')) \\
\downarrow & \downarrow \text{Lan}_F(X)(F(a)) & \downarrow \text{Lan}_F(X)(F(a)) \\
X(A') & \xrightarrow{\lambda^F(A')_{(A', \text{id}_F(A'))}} & \text{Lan}_F(X)(F(A'))
\end{array}
\]
while the definition of $\text{Lan}_F(X)$ gives

\[
\begin{array}{c}
X(A) \\
\downarrow_{\lambda^{F(A)}_{(A,F(a))}} \\
\text{Lan}_F(X)(F(A)) \\
\downarrow_{\text{Lan}_F(X)(F(a))} \\
\text{Lan}_F(X)(F(A'))
\end{array}
\]

hence,

\[
\begin{array}{c}
X(A) \\
\downarrow_{\eta_{A}} \\
\text{Lan}_F(X)(F(A)) \\
\downarrow_{\text{Lan}_F(X)(F(a))} \\
\text{Lan}_F(X)(F(A')) .
\end{array}
\]

Given $Y : B \to C$ and $\gamma : X \Rightarrow Y \circ F$, $B \in |B|$ implies $(Y(B), \{Y(\beta) \circ \gamma_A \mid (A, \beta) \in |F \downarrow B|\})$ is a cocone on $X \circ F B : F \downarrow B \to C$: for $a \in A(A, A')$, naturality of $\gamma$ implies

\[
\begin{array}{c}
X(A) \\
\downarrow_{\eta_{A}} \\
\text{Lan}_F(X)(F(A)) \\
\downarrow_{\text{Lan}_F(X)(F(a))} \\
\text{Lan}_F(X)(F(A'))
\end{array}
\]

hence

\[
\begin{array}{c}
F(A) \\
\downarrow_{\beta} \\
B \\
\downarrow_{\beta'} \\
F(A')
\end{array}
\]

implies

\[
Y(\beta') \circ \gamma_{A'} \circ X(a) = Y(\beta') \circ Y(F(a)) \circ \gamma_A \\
= Y(\beta' \circ F(a)) \circ \gamma_A \\
= Y(\beta) \circ \gamma_A
\]

as depicted in

\[
\begin{array}{c}
X(A) \\
\downarrow_{\eta_{A}} \\
Y(B) \\
\downarrow_{Y(\beta') \circ \gamma_{A'}} \\
X(A') .
\end{array}
\]

Let $\tilde{\gamma}_B : \text{Lan}_F(X)(B) \to Y(B)$ be the unique induced morphism

\[
\begin{array}{c}
X(A) \\
\downarrow_{\lambda^B_{(A,\beta)}} \\
\text{Lan}_F(X)(B) \\
\downarrow_{\tilde{\gamma}_B} \\
Y(B)
\end{array}
\]
These components give a natural transformation \( \tilde{\gamma} : \text{Lan}_F(X) \Rightarrow Y \): for \((A, \beta) \in |F \downarrow B|\), the definition of \( \text{Lan}_F(X) \) justifies the first equality of

\[
\tilde{\gamma}_{B'} \circ \text{Lan}_F(X)(b) \circ \lambda^B_{(A, \beta)} = \gamma_{B'} \circ \lambda^B_{(A, b \circ \beta)} = Y(b \circ \beta) \circ \gamma_A = Y(b) \circ Y(\beta) \circ \gamma_A = Y(b) \circ \tilde{\gamma}_B \circ \lambda^B_{(A, \beta)}
\]

while the definition of \( \tilde{\gamma} \) justifies the second and fourth. These imply \( \tilde{\gamma}_{B'} \circ \text{Lan}_F(X)(b) = Y(b) \circ \tilde{\gamma}_B \) as depicted in

\[
\begin{array}{ccc}
\text{Lan}_F(X)(B) & \xrightarrow{\tilde{\gamma}_B} & Y(B) \\
\text{Lan}_F(X)(b) & \downarrow & Y(b) \\
\text{Lan}_F(X)(B') & \xrightarrow{\tilde{\gamma}_{B'}} & Y(B').
\end{array}
\]

Moreover, \( \tilde{\gamma} \) satisfies

\[
[(\tilde{\gamma} \ast id_F) \circ \eta]_A = \tilde{\gamma}_{F(A)} \circ \eta_A = \tilde{\gamma}_{F(A)} \circ \lambda^{F(A)}_{(A, id_{F(A)})} = Y(id_{F(A)}) \circ \gamma_A = \gamma_A
\]

This determines \( \tilde{\gamma} \) uniquely: if \( \psi : \text{Lan}_F(X) \Rightarrow Y \) satisfies \((\psi \ast id_F) \circ \eta = \gamma\) then \( A \in |A| \) implies \( \gamma_A = \psi_{F(A)} \circ \eta_A = \psi_{F(A)} \circ \lambda^{F(A)}_{(A, id_{F(A)})} \). \( A, \beta) \in |F \downarrow B| \) implies

\[
\tilde{\gamma}_B \circ \lambda^B_{(A, \beta)} = Y(\beta) \circ \gamma_A = Y(\beta) \circ \psi_{F(A)} \circ \lambda^{F(A)}_{(A, id_{F(A)})} = \psi_B \circ \text{Lan}_F(X)(\beta) \circ \lambda^{F(A)}_{(A, id_{F(A)})} = \psi_B \circ \lambda^B_{(A, \beta)}
\]

where the definition of \( \tilde{\gamma} \) justifies the first equality, naturality of \( \psi \) justifies the third, and the definition of \( \text{Lan}_F \) gives the last. Together these imply \( \tilde{\gamma} = \psi \).

\[2.825\] Let \( F : A \to B \) and \( X : A \to C \) be functors with \( A \) and \( B \) small. \( C^F : C^B \to C^A \) has a left adjoint if \( C \) admits certain colimits: specifically for each \( B \in |B| \), \( C \) admits a colimit of

\[
B \downarrow F \xrightarrow{B_F} A \xrightarrow{X} C.
\]

\[2.826\] \( C \) admits a colimit of \( X : A \to C \) iff there exists a left Kan extension of \( X \) along \( A \to 1 \).

Because: \( \Rightarrow \): A functor \( L : 1 \to C \) is a \( C \)-object and a natural transformation \( X \to L \circ ! \) is a cocone on \( X \).

\( \Leftarrow \): the one comma category of interest is isomorphic to \( A \).

\[\blacksquare\]
Chapter 3

Categories Composed of Functions

Mathematical structures such as groups and dynamic systems are typically built from sets. The Zermelo-Fraenkel axioms name the legal constructions with sets and many categorical definitions are abstracted from these operations. Moreover, the classical Kan extensions algorithms for which we have developed algorithms involve \textit{Set}-valued functors (referred to as \textit{actions}).

3.1 The Category Composed of Functions

Assume a model of the axioms of set theory. See Appendix A or reference \cite{121}.

\( f, g : X \to Y \) are equal if \( \forall x \ f(x) = g(x) \). That is, any one-point set is a generator in \textit{Set}.

3.11 Special Morphisms. \( f : X \to Y \) in set is \underline{injective} if \( f(x) = f(x') \) implies \( x = x' \). It is \underline{surjective} if for each \( y \in Y \) there is \( x \in X \) for which \( y = f(x) \). A function with both properties is \underline{bijective}.

3.111 \( X \to Y \) in \textit{Set} is \underline{monic} iff injective.

\( \Rightarrow \): Let \( x_1, x_2 \in X \) induce \( \hat{x}_i : \{\ast\} \to X \) via \( \hat{x}_i(\ast) = x_i \). \( m(\hat{x}_1(\ast)) = m(\hat{x}_2(\ast)) \) implies \( m \circ \hat{x}_1 = m \circ \hat{x}_2 \). \( m \) monic yields \( \hat{x}_1 = \hat{x}_2 \), hence, \( x_1 = x_2 \).

\( \Leftarrow \): Given \( W \xrightarrow{\hat{m}} X \xrightarrow{\hat{f}} Y \) implies \( \forall_w m(f(w)) = m(g(w)) \) implies \( \forall_w f(w) = g(w) \), hence, \( f = g \).

3.112 \( X \to Y \) in \textit{Set} is \underline{epic} iff surjective.

\( \Rightarrow \): Surjectivity holds without the condition on \( m : X \to Y \) if \( Y = \phi \) so assume \( m \) epic and \( Y \neq \phi \). Let \( 2 = \{0, 1\} \) and define \( \theta : Y \to 2 \) by \( y \mapsto 0 \). For \( \bar{y} \in Y \) define \( \chi_{\bar{y}} : Y \to 2 \) by

\[
\chi_{\bar{y}}(y) = \begin{cases} 1 & \text{if } y = \bar{y} \\ 0 & \text{otherwise} \end{cases}
\]

\( \bar{y} \in Y \) and \( \forall x \ \bar{y} \neq m(x) \) implies \( \chi_{\bar{y}} \neq \theta \) and \( \chi_{\bar{y}} \circ m = \theta \circ m \), contradicting the hypothesis on \( m \).

\( \Leftarrow \): Given \( W \xrightarrow{\hat{m}} X \xrightarrow{\hat{f}} Y \), \( X = \phi \) implies \( f = g \). \( x \in X \) implies \( \exists_w m(w) = x \) implies \( f(x) = g(x) \).
\[ f(m(w)) = g(m(w)) = g(x) \text{ implies } f = g. \]

3.113 \( X \to Y \text{ in } \textbf{Set} \) is iso iff bijective iff monic and epic.

Because: \( \Rightarrow \): an iso is monic and epic in any category. In \textbf{Set} this implies injective and surjective, hence, bijective.

\( \Leftarrow \): \( f : X \to Y \) epic implies existence of \( g : Y \to X \) for which \( \forall y, f(g(y)) = y \), hence, \( f \circ g = \text{id}_Y \). In particular, \( \forall x, f(g(f(x))) = f(x) \). \( f \) monic implies \( \forall x, g(f(x)) = x \), hence, \( g \circ f = \text{id}_X \).

For \( f \in \text{Set}(X, Y) \) and \( V \subset Y \), \( f^* \) denotes the inverse image of \( V \) under \( f \):
\[ f^*(V) = \{ x \in X \mid f(x) \in V \}. \]

3.12 Products. \textbf{Set} has small products.

Because: for a collection \( \{ X_\alpha \mid \alpha \in A \} \) of sets the cartesian product \( \prod X_\alpha \) may be constructed via the axioms of Zermelo-Fraenkel set theory [69] (union gives \( \bigcup X_\alpha \), pairing gives \( A \times \bigcup X_\alpha \), comprehension gives the subset consisting of functions \( f : A \to \bigcup X_\alpha \) for which \( f(\alpha) \in X_\alpha \)). Together with the projection maps \( \prod X_\alpha \xrightarrow{\pi_\beta} X_\beta \) defined by \( (x_\alpha) \mapsto x_\beta \) this gives a product in \textbf{Set}. Given \( X_\beta \)
\[ f : Z \to \prod X_\alpha \text{ for which } \forall \alpha, \pi_\alpha \circ f = f_\alpha \text{ is } z \mapsto (f_\alpha(z)). \]

3.121 \textbf{Set} has equalizers.

Because: from \( X \xrightarrow{g} Y \) comprehension gives \( X' = \{ x \in X \mid f(x) = g(x) \} \). With \( e : X' \to X \) inclusion we have \( X' \xrightarrow{e} X \xrightarrow{f} Y \). Given \( X' \xrightarrow{e} X \xrightarrow{f} Y \) define \( \hat{h} : Z \to X' \) by \( z \mapsto h(z) \). \( f \circ \hat{h} = g \circ h \)
\[ \text{implies } \forall z, f(h(z)) = g(h(z)) \text{ implies } \hat{h} \text{ well-defined. Moreover } X' \xrightarrow{e} X \xrightarrow{f} Y. \text{ } e \circ \psi = h \text{ implies } \]
\[ \forall z, \psi(z) = e(\psi(z)) = h(z) = \hat{h}(z) \text{ implies } \psi = \hat{h}. \]

3.122 \textbf{Set} is complete, hence, cartesian.

Because: it has equalizers and small products.
3.13 Coproducts. Set has small coproducts.

Because: for a collection \( \{X_\alpha \mid \alpha \in A\} \) of sets, ZF permits construction of \( \{(x, \alpha) \mid \alpha \in A, x \in X_\alpha\} = \bigcup (X_\alpha \times \{\alpha\}) \). Denote this set by \( \prod X_\alpha \). Together with the \( X_\beta \xrightarrow{\lambda_\beta} \prod X_\alpha \) via \( x \mapsto (x, \beta) \) this gives a coproduct in Set. Given \( (x, \alpha) \mapsto f_\alpha(x) \).

A coproduct of \( X = \{a, b\} \) with itself, for example, is

\[
\{a, b\} \xrightarrow{\lambda_1} \{(a, 1), (b, 1)\} \xrightarrow{\lambda_2} \{a, b\}
\]

where \( \lambda_i(x) = (x, i) \).

3.131 Set has coequalizers.

Because: from \( X \xrightarrow{f+g} Y \) comprehension gives the binary relation

\[
Y' = \{(f(x), g(x)) \in Y \times Y \mid x \in X\}
\]
on \( Y \). Existence of the equivalence relation generated by \( Y' \) is established by Zorn’s lemma \( \mathcal{P}(\mathcal{P}(Y \times Y)) \) is partially ordered by containment \( \subseteq \). Let \( \Phi_{f,g} \) be its subposet of equivalence relations containing \( Y' \). \( \mathcal{P}(Y \times Y) \in \Phi_{f,g} \) implies \( \Phi_{f,g} \) nonempty. We must show that any \( E_1 \supset E_2 \supset \cdots \Phi_{f,g} \) has an upper bound in \( \Phi_{f,g} \). \( E = \bigcap E_i \) implies \( E_i \supset E \). To show \( E \) an equivalence relation: \( y \in Y \) implies \( \forall_i (y, y) \in E_i \) implies \( (y, y) \in E \); \( (y, y') \in E \) implies \( \forall_i (y, y') \in E_i \) implies \( \forall_i (y', y) \in E_i \) implies \( (y', y') \in E \); \( (f(x), g(x)) \in \bigcap E_i \) \( \forall_i (f(x), g(x)) \in E_i \) implies \( (f(x), g(x)) \in E \) so \( Y' \subset E \) and \( E \in \Phi_{f,g} \).

Let \( (Y/E) \) be the set of equivalence classes of \( Y \) under the \( E \). For \( y \in Y \) let \([y]\) denote its equivalence class. Define \( \xi : Y \to (Y/E) \) by \( y \mapsto [y] \). \( x \in X \) implies \( (f(x), g(x)) \in E \), \( [f(x)] = [g(x)] \), and implies
\[ \exists(f(x)) = \exists(g(x)), \text{ hence, } X \xleftarrow{f} \xrightarrow{g} Y \xrightarrow{j} (Y/E). \text{ Given } X \xleftarrow{f} \xrightarrow{g} Y \xrightarrow{h} Z \text{ define a binary relation } \]

\[ H \text{ on } Y \text{ by } (y, y') \in H \text{ iff } h(y) = h(y'). H \text{ is an equivalence relation: } h(y) = y(y); h(y) = h(y') \implies h(y') = h(y); h(y) = h(y') \text{ and } h(y') = h(y'') \implies h(y) = h(y''). \text{ It contains } Y'': \forall_x h(f(x)) = h(g(x)), \text{ hence, contains } E. \text{ Define } \hat{h} : (Y/E) \to Z \text{ by } [y] \mapsto h(y). [y] = [y'] \implies (y, y') \in H \implies h(y) = h(y') \text{ so } \hat{h} \text{ is well defined. Moreover, } \hat{h} \circ \exists = h. \]

To show uniqueness: \( \theta \circ \exists = h \) implies \( \theta([y]) = \theta(\exists(y)) = h(y) = \hat{h}([y]) \) so \( \theta = \hat{h} \).

Donald E. Knuth describes an algorithm for computing the equivalence relation on a finite set generated by a binary relation on that set (see page 354 of [96]). Barr and Wells [17] outline another algorithm and refer to [117].

### 3.132 Set is cocomplete.

*Because:* it admits coequalizers and small coproducts.

### 3.133 A pushout of

\[
\begin{array}{ccc}
X & \xrightarrow{g} & Y \\
\downarrow{f} & & \downarrow{\exists} \\
Y & \xrightarrow{\lambda} & Z
\end{array}
\]

in Set is given by

\[
\begin{array}{ccc}
X & \xrightarrow{g} & Z \\
\downarrow{f} & & \downarrow{\lambda'} \\
Y & \xrightarrow{\lambda} & Q
\end{array}
\]

where \( Q = (Y \bigsqcup Z)/\sim, Y \bigsqcup Z = \{(y, 0) \mid y \in Y\} \cup \{(z, 1) \mid z \in Z\} \) and \( \sim \) is the equivalence relation on \( Y \bigsqcup Z \) generated by: \{insert formula for \( \sim\}, \lambda(y) = [(y, 0)], \text{ and } \lambda'(z) = [(z, 1)].\)

### 3.14 Images. Set has images.

*Because:* if \( f \in \text{Set}(X, Y) \) and

\[ Y' = \{y \in Y \mid (\exists x \in X). (y = f(x))\}, \]

then \( i : Y' \to Y \) defined by \( i(y) = y \) is injective, hence, monic by 3.111. \( i \) allows \( f \) since \( f = i \circ f | \) where \( f | : X \to Y' \) is defined by \( f_1(x) = f(x) \).

Suppose \( j : Y'' \to Y \) is monic and \( f = j \circ \hat{f} \) for some \( \hat{f} : X \to Y'' \). Define \( m : Y' \to Y'' \) by \( m(y) = \hat{f}(x) \) where \( y = f(x) \). To see that \( m \) is well-defined, assume \( y = f(x) = f(x') \). Then \( j(\hat{f}(x)) = j(\hat{f}(x')) \). \( j \) monic implies \( f(x) = f(x') \).
3.2 The Category of Finite Sets

Let $\textbf{Set}_f$ be the full subcategory of $\textbf{Set}$ having the finite sets as objects. Monos, epis, and isos are as in $\textbf{Set}$. $\textbf{Set}_f$ has all finite limits and colimits. $\textbf{Set}_f$ is a topos, hence, is cartesian, regular, coherent, and Heyting. $\textbf{Set}_f$ is equivalent to its full subcategory, $\Delta$, having all sets of the form $\underline{n} = \{1, 2, \ldots, n\}$ and $\underline{0} = \emptyset$ as objects. $\Delta$ is the simplicial category. See [102] for details.
Chapter 4

Groups

In this chapter we review concepts and results from group theory. A group can be construed as a category that has a single object: the group elements are the morphisms and the group operation is composition. Moreover, group homomorphisms coincide with functors between such categories. These facts can be usefully applied to generalize concepts from group theory to other domains. One example is the Carmody-Walters algorithm for computing Kan extensions which is an adaptation of the Todd-Coxeter coset-enumeration procedure from group theory.

4.1 Groups and Homomorphisms

A group \((G, *, e)\) consists of

- a set \(G\),
- a function \(* : G \times G \to G\), and
- an element \(e \in G\)

These ingredients are subject to the following axioms

- * is associative: for all \(x, y, z \in G\), \((x * (y * z)) = ((x * y) * z)\)
- \(e\) is a *-identity: for all \(x \in G\), \(e * x = x = x * e\)
- inverses exist: for all \(x \in G\), there is an \(x^{-1} \in G\) for which \(x * x^{-1} = e = x^{-1} * x\).

For brevity, we may refer to a group \((G, *, e)\) simply as \(G\) in which case the group operation * and identity element \(e\) must be inferred from context. In cases where we seek to be more explicit, we let \(U(G)\) denote the underlying set of the group.

A group is **abelian** if: for all \(x, y \in G\), \(x * y = y * x\).

4.11 Subgroups. A subset \(H\) of a group \(G\) is a **subgroup** of \(G\) if

- for all \(x \in H\), \(x^{-1} \in H\),
- for all \(x, y \in H\), \(x * y \in H\).
These conditions imply that \( e \in H \). The group operation on \( H \) is the restriction of \( * \) to a function \( H \times H \to H \) which is well-defined by the second condition in the definition of subgroup.

### 4.12 Cosets

Let \( G \) be a group and let \( H < G \) be a subgroup. A subset \( S \subseteq G \) is a left coset of \( H \) in \( G \) if there is an element \( g \in G \) for which \( S = \{ g \ast h \in G \mid h \in H \} \) in which case we write \( S = gH \). An element \( g \in G \) which generates \( S \) in this way is a representative of the coset. Each element \( gh \) with \( g \in G \) and \( h \in H \) is a representative of \( gH \).

A subset \( S \subseteq G \) is a right coset of \( H \) in \( G \) if there is an element \( g \in G \) for which \( S = \{ h \ast g \in G \mid h \in H \} \) in which case we write \( S = Hg \). The definition of representatives of right cosets is analogous to that for left cosets.

#### 4.121

If \( H \) is a subgroup of a group \( G \) and let \( x, y \in G \). The following are equivalent.

i) \( xH = yH \)

ii) \( y^{-1}x \in H \)

iii) \( x^{-1}y \in H \)

Moreover, the following are equivalent.

iv) \( xy^{-1} \in H \)

v) \( yx^{-1} \in H \)

vi) \( Hx = Hy \)

*Because:* i) \( \iff \) ii) and vi) \( \iff \) v) are Lemma 2.8 in [116]. ii) \( \iff \) iii) holds since \( H \) is a subgroup and \( (y^{-1}x)^{-1} = x^{-1}y \). Similarly, iv) \( \iff \) v) holds since \( H \) is a subgroup and \( (yx^{-1})^{-1} = xy^{-1} \).

#### 4.122

In general, neither ii) nor iii) implies iv) or v) in 4.121. Consider, for example, the symmetric group \( S_3 \) with \( a = (1 \ 2) \), \( b = (1 \ 3) \), \( c = (2 \ 3) \), \( d = ab = (1 \ 3 \ 2) \) and \( e = ba = (1 \ 2 \ 3) \). Its multiplication table is given below.

<table>
<thead>
<tr>
<th>x ( \ast ) y</th>
<th>1</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>e</td>
</tr>
<tr>
<td>a</td>
<td>a</td>
<td>1</td>
<td>d</td>
<td>e</td>
<td>b</td>
<td>c</td>
</tr>
<tr>
<td>b</td>
<td>b</td>
<td>e</td>
<td>1</td>
<td>b</td>
<td>c</td>
<td>a</td>
</tr>
<tr>
<td>c</td>
<td>c</td>
<td>d</td>
<td>e</td>
<td>1</td>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>d</td>
<td>d</td>
<td>c</td>
<td>a</td>
<td>b</td>
<td>e</td>
<td>1</td>
</tr>
<tr>
<td>e</td>
<td>e</td>
<td>b</td>
<td>c</td>
<td>a</td>
<td>1</td>
<td>d</td>
</tr>
</tbody>
</table>

\( H = \{1, a\} \) is a subgroup. Its left cosets are \( H \), \( bH = cH = \{b, e\} \) and \( cH = dH = \{c, d\} \). Its right cosets are \( H \), \( Hb = Hd = \{b, d\} \) and \( Hc = He = \{c, e\} \). In particular, \( e^{-1}b = bd = a \in H \) but \( b^{-1}e = bd = c \notin H \). Similarly, \( b^{-1}e = be = a \in H \) but \( eb^{-1} = eb = c \notin H \).

#### 4.123

Let \( H \) be a subgroup of a group \( G \). The set of left cosets of \( H \) form a partition of \( G \) as do the set of right cosets.

*Because:* this is Theorem 2.9 in [116].

#### 4.124

Let \( H \) be a subgroup of a group \( H \). The **index** of \( H \) in \( G \) is the number of left cosets \( gH \) and is denoted by \( [G : H] \). The set of left cosets is denoted \( G/H \). In particular, \( |G/H| = [G : H] \). As a consequence of 4.125, we do not need to define the index for right cosets.
In certain cases (i.e., if $K$ is a normal subgroup of $G$ and, in particular, if $G$ is an abelian group), $G/K$ will inherit a group structure from $G$.

4.125 Let $H$ be a subgroup of a group $H$. The number of left cosets of $H$ in $G$ is equal to the number of right cosets.

*Because:* this is Theorem 2.10 of [116]. In particular, if $R$ and $L$ are the sets of right and left cosets, then the function $f : R \rightarrow L$ defined by $f(Hg) = g^{-1}H$ is a bijection.

### 4.13 Homomorphisms

Let $(G, \ast, e)$ and $(H, \star, u)$ be groups. A function $f : G \rightarrow H$ is a homomorphism if $f(e) = u$ and for all $x, y \in G$, $f(x \ast y) = f(x) \star f(y)$.

*Because:* groups and group homomorphisms form a category. A composite of group homomorphisms is a homomorphism, composition is associative, and identity functions are homomorphisms.

4.131 The free group construction induces a left adjoint $F : \text{Set} \rightarrow \text{Group}$ to the underlying set functor $U : \text{Group} \rightarrow \text{Set}$.

*Because:* the definition of free group implies that there is a bijection $\varphi : \text{Group}(F(x), G) \Rightarrow \text{Set}(X, U(G))$. Naturality of this bijection is a simple calculation.

### 4.14 Group Presentations

Let $X$ be a set and let $X^{-1}$ be a set that is disjoint from $X$ and for which there is a bijection $X \rightarrow X^{-1}$, which we denote by $x \mapsto x^{-1}$. Let $X'$ be a singleton set that is disjoint from $X \cup X'$. We denote the lone element of $X'$ by 1.

Let $X$ be a set. A word on $X$ is a sequence $w = (a_1, a_2, \ldots)$ where $a_i \in X \cup X^{-1} \cup X'$ and $a_k = 1$ for all $k \geq N$ for some $N \in \mathbb{N}$. The word $(1, 1, \ldots)$ is empty. The smallest integer $N$ for which $a_k = 1$ for all $k > N$ is the length of the word.

4.141 If $X$ is any set, then the set of words on $X$ form a group.

*Because:* this is Theorem 11.1 of [116].

The group constructed in the proof of 4.141 is the free group on $X$.

4.142 The adjunction of 4.132 asserts that given any function $f : X \rightarrow U(G)$ from a set $X$ to the underlying set of a group $G$, there is a unique homomorphism from $F(X)$ to $G$ that extends $f$. 

\[
\begin{array}{c}
F(X) \\
\downarrow \\
U(F(X)) \\
\end{array}
\xrightarrow{\eta} G \xrightarrow{f} U(G) \xrightarrow{\eta} F(X)
\]
4.143 Every group is a quotient of a free group.

Because: this is Corollary 11.2 of [116].

4.144 Let $X$ be a set. A relation (or an $X$-relation) is pair of words $(w, w')$ on $X$ (see 4.14). A presentation $P$ of a group is a pair $(X, \mathcal{R})$ where $X$ is a set and $\mathcal{R}$ is a set of words on $X$. The group presented by $P$ is constructed as follows. First form the free group $F(X)$ generated by $X$. The relations define a binary relation $R$ on the elements of $F(X)$. The group generated by $P$ is $F(X)/N$ where $N$ is the normal subgroup generated by $R$. See [116].

4.2 Todd-Coxeter Coset Enumeration Procedure

Given a finite presentations of a group $G$ and a subgroup $H$ of $G$, the Todd-Coxeter procedure is an algorithm for finding the index of $H$ in $G$ if that index is finite. In particular, if $H$ is the trivial subgroup, the procedure finds the order of $G$ if it is finite.

Coset enumeration has been used to investigate finitely-presented groups for over 100 years [105]. The Todd-Coxeter algorithm was introduced in 1936 in [122]. Since its discovery, the Todd-Coxeter algorithm has been used extensively to guide manual calculations. The first computer implementation was in 1953 by Haselgrove [77] and the first published description of an implementation was in 1961 in [68]. [101] describes the early history of such implementations. In [35] and later in [34], the algorithm is described as the most widely used computational technique in group theory.

4.21 The Algorithm. We describe the algorithm in a manner that parallels are description of the Carmody-Walters algorithm for computing left Kan extensions in 6.2. Let $(X, |\mathcal{R}|)$ be a finite presentation of a group. The Todd-Coxeter Procedure involved constructing three types of tables. There is a single $\epsilon$ table, a single $L$-table and a set relation tables. In the Todd-Coxeter procedure, the $\epsilon$-table is trivial: it has two columns and one row which is completed early on in the calculation and ignored thereafter. The $L$- and relation-tables are more interesting. Each has a fixed number of columns (although different tables may have different numbers of columns) and an unknown but nonzero number of rows. Initially, the entries of all tables except the $\epsilon$-table are blank (or undefined). As the algorithm proceeds, we successively add entries $0, 1, 2, \ldots$ into the $L$-tables, fill in consequences of those new entries in the relation tables.

The $\epsilon$-table is as follows.

<table>
<thead>
<tr>
<th>$X(\ast)$</th>
<th>$L(F(\ast))$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0$</td>
<td></td>
</tr>
</tbody>
</table>

If the presentation has $n$ generators (i.e., $|X| = n$), then The $L$-table has $2n + 1$ columns as shown below. It has a first column and, for each generator $x$, it has one column for $x$ and and for $x^{-1}$.

<table>
<thead>
<tr>
<th>$L(x)$</th>
<th>$L(x^{-1})$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L(\ast)$</td>
<td>$L(\ast)$</td>
</tr>
</tbody>
</table>

We leave the bottom of each column open since the number of rows is not known a priori.

For each relation $r \in \mathcal{R}$ of the group presentation, we construct a relation table as follows. If $r$ is $x_1 \cdots x_m = y_1 \cdots y_n$, then the corresponding table has $m + n + 2$ columns.
As in the $L$-table, we leave the bottom of each column of the relation tables open since the number of rows is not known a priori. In the special case of a relation $x_1 \cdots x_m = 1$, we construct the relation table as follows.

The pseudo-code shown in Listing 1 sketches the algorithm. The manner in which one fills in the details of this sketch leads to distinct implementations and can have a dramatic effect on the performance of the algorithm in some cases. In line 3, for example, we may (1) define a single new entry in the $L$-table, or (2) define enough entries to complete a row of the $L$-table or (3) make some other choice.

[Listing 1: Pseudo-code for the Todd-Coxeter algorithm]

4.22 Complexity of the Algorithm. The following metrics are defined in [35] to characterize the complexity of Todd-Coxeter Procedure calculations.

\[ I = \text{index of } H \text{ in } G \]
\[ M = \text{maximum number of cosets defined at any instant during the enumeration} \]
\[ T = \text{total number of cosets defined during the enumeration} \]
\[ t = \text{execution time} \]
\[ \tau = M/I \]
\[ \epsilon = T/I \]

The subscripts $F$ and $H$ applied to the metrics are used to distinguish values computed using the Felsch and Haselgrove variants of Todd-Coxeter [35]. Table 4.1 shows a small sample of the metrics calculated for particular coset enumeration problems in [35]. Notice, for example, the subgroup $\langle ab, c \rangle$ generates Coxeter group $G_{3,7,17}^H$ (hence, has one coset) but over 1,000 cosets must be computed in order to determine that.
The influential paper [35] describes finite presentations of the trivial group that require arbitrarily many cosets to be generated during execution of the Todd-Coxeter Procedure. This suggests that characterizing the complexity of the algorithm may be challenging. A more serious challenge is implied by the unsolvability of the Word Problem. That is, no Turing machine (or equivalent computational model under the Church-Turing Thesis), can determine whether an arbitrary word in the generators of a finitely-presented group is equivalent to the identity element. This fact was proved independently by P. S. Novikov (1955), W. W. Boone (1954–1957) and J. L. Britton (1958). The word problem for groups was first considered by M. Dehn (1910) and A. Thue (1914) [116].

4.3 The Abelian Stabilizer Problem

The influential paper [95] by Kitaev gave a polynomial quantum algorithm for the Abelian Stabilizer Problem. In this section we show how the problem may be described as a Kan extension.

4.31 Group Actions. A group $G$ may be construed as a category with a single object that we will denote by $\ast$. Morphisms of the category are the group elements, the identity morphism is the identity group element and composition is given by the group operation. A functor $X : G \to \text{Set}$ is a (left) group action [116]. In particular, $X(\ast)$ is a set and, for each $g \in G$, $X(g) : X(\ast) \to X(\ast)$ is a function. The definition of functor implies that $X(e)$ is the identity function on $X(\ast)$ and, if $g, g' \in G$, then $X(g) \circ X(g') = X(g \circ g')$. Note that a right group action may be defined as a contravariant functor.

4.32 Stabilizers. A stabilizer of $x \in X(\ast)$ is $G_x = \{ g \in G \mid X(g)(x) = x \}$ (using the notation of [116]). In other words, $G_x$ is the set of $G$-elements that leave $x$ fixed. It is easy to show that $G_x \triangleleft G$ is a subgroup. If $G_x \triangleleft G$ is a normal subgroup, in particular, if $G$ is abelian, then the natural map $\pi : G \to G/G_x$ is a (surjective) group homomorphism, hence, a functor.

<table>
<thead>
<tr>
<th>Group</th>
<th>Presentation</th>
<th>$I$</th>
<th>$M_F$</th>
<th>$M_H$</th>
<th>$T_F$</th>
<th>$T_H$</th>
<th>$\tau_F$</th>
<th>$\tau_H$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$G^{3,7,17}/E$</td>
<td>$a^3 = b^7 = c^{17} = (ab)^2 =$</td>
<td>504</td>
<td>504</td>
<td>775</td>
<td>504</td>
<td>1,222</td>
<td>1.00</td>
<td>1.54</td>
</tr>
<tr>
<td></td>
<td>$(bc)^2 = (ca)^2 = (abc)^2 = 1$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\text{PSL}_3(4) \mid \langle a \rangle$</td>
<td>$a^5 = b^3 = (ab)^4 =$</td>
<td>4,032</td>
<td>4,032</td>
<td>30,537</td>
<td>4,655</td>
<td>34,439</td>
<td>1.00</td>
<td>7.57</td>
</tr>
<tr>
<td></td>
<td>$(a^{-1} b^{-1} a^{-1} a b)^3 =$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$b a^{-2} b^{-1} a^{-1} b^{-1} a b^{-1} a^{-1} = 1$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Weyl $B_6 \mid E$</td>
<td>$a^2 = b^2 = c^2 = d^2 = e^2 = f^2 =$</td>
<td>46,080</td>
<td>46,080</td>
<td>46,094</td>
<td>46,080</td>
<td>74,382</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td></td>
<td>$(ab)^3 = (ac)^2 = (ad)^2 = (ae)^2 =$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$(af)^2 = (bc)^3 = (bd)^2 = (be)^2 =$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$(de)^3 = (df)^2 = (ef)^4 = 1$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$G^{3,7,17}/ \langle ab, c \rangle$</td>
<td>$a^3 = b^7 = c^{17} = (ab)^2 =$</td>
<td>1</td>
<td>1,471</td>
<td>2,764</td>
<td>1,471</td>
<td>3,903</td>
<td>1,471</td>
<td>2,764</td>
</tr>
<tr>
<td></td>
<td>$(bc)^2 = (ca)^2 = (abc)^2 = 1$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\text{PSL}_2(11) \mid E$</td>
<td>$a^{11} = b^2 = (ab)^3 = (a^4 ba^{-5} b)^2 = 1$</td>
<td>660</td>
<td>1,066</td>
<td>1,188</td>
<td>1,118</td>
<td>1,495</td>
<td>1.61</td>
<td>1.80</td>
</tr>
</tbody>
</table>

Table 4.1: Complexity metrics calculated for applications of the Todd-Coxeter Procedure [35]
4.33 **Orbits.** Let $G$ be a group and let $X : G \to \textbf{Set}$ be a group action. An orbit of $y \in X(*)$ is $\mathcal{O}(y) = \{X(g)(y) \mid g \in G\}$. In other words, the orbit of $y$ is the set of images of $y$ under the action $X$.

4.331 **Distinct group orbits are disjoint.** That is, the orbits form a partition of $X(*)$.

Because: $z \in \mathcal{O}(y) \cap \mathcal{O}(y')$ implies that $z = X(g)(y)$ for some $g \in G$ which implies that $X(g^{-1})(z) = X(g^{-1})(X(g)(y)) = y$.

4.34 **Kan Extension.** In order to formulate the Abelian Stabilizer Problem as a Kan extension, we need a diagram of the following form where $\epsilon : X \to X' \circ \natural$ is a natural transformation.

\[
\begin{array}{ccc}
G & \xrightarrow{\natural} & G/G_x \\
\downarrow \rho & & \downarrow \gamma \\
Set & \xrightarrow{\eta} & X'
\end{array}
\]

4.341 Let $G$ be an abelian group, let $X : G \to \textbf{Set}$ be a group action and let $G_x \triangleleft G$ be the stabilizer of an element $x \in X(*)$. Define $X' : G/G_x \to \textbf{Set}$ by

\[X'(*) = \{\mathcal{O}(y) \mid y \in X(*)\} \text{ and } X'(g \, G_x)(\mathcal{O}(y)) = \mathcal{O}(X(g)(y)) \, .\]

Then $X'$ is a functor.

Because: $X'(*)$ is a well-defined set. For a coset $g \, G_x$ we must show that $X'(g \, G_x) : X'(*) \to X'(*)$ is a well-defined function. Suppose $\mathcal{O}(y) = \mathcal{O}(y')$. We must show that $\mathcal{O}(X(g)(y)) = \mathcal{O}(X(g)(y'))$. By hypothesis, $y = X(h)(y')$ for some $h \in G$. If $z \in \mathcal{O}(X(g)(y))$, then, for some $u \in G$, $z = X(u)(X(g)(y)) = X(u \, g)(y) = X(u \, g)(X(h)(y')) = X(u \, g \, h)(y') = X(u \, h \, g)(y') \in \mathcal{O}(X(g)(y'))$. The conclusion follows from the fact that orbits form a partition of $X(*)$.

4.342 Let $G$, $X$ and $X'$ be as in 4.341. Define $\epsilon_* : X(*) \to X'(*)$ by $\epsilon_*(y) = \mathcal{O}(y)$. Then $\epsilon : X \Rightarrow X' \circ \natural$ is a natural transformation.

Because: commutativity of

\[
\begin{array}{ccc}
X(*) & \xrightarrow{\epsilon_*} & X'(*) \\
\downarrow \delta & & \downarrow \chi \\
X(g) & \xrightarrow{\rho} & X'(g \, G_x) \\
\end{array}
\]

is a straightforward calculation.

4.343 In the diagram of 4.34, $(X', \epsilon)$ is a left Kan extension of $X$ along $\natural$.

Because: this follows from maximality of $G_x$.

The challenge of the Abelian Stabilizer Problem, however, is not to find the Kan extension $(X', \epsilon)$ once
we know $G_x$, it is to find the stabilizer $G_x$ of a given $x \in X(*)$. Once $G_x$ is known, the Kan extension is given by the equations above.

4.4 The Hidden Subgroup Problem

Many quantum algorithms that offer superior performance over corresponding classical algorithms are instances of the Hidden Subgroup Problem. In this section we define the problem and describe particular cases of it.

### 4.4.1 Constant Functions on Cosets

Let $H$ be a subgroup of a group $G$ and let $X$ be a set. A function $f : U(G) \to X$ is constant on left cosets of $H$ if: for all $x, y \in G$, if $xH = yH$, then $f(x) = f(y)$. Similarly, $f$ is constant on right cosets of $H$ if: for all $x, y \in G$, if $Hx = Hy$, then $f(x) = f(y)$.

#### 4.4.1.1

Let $H$ be a subgroup of a group $G$, let $X$ be a set and let $f : U(G) \to X$ be a function. The following are equivalent:

- $f$ is constant on left cosets of $H$,
- for all $x, y \in G$, if $y^{-1}x \in H$, then $f(x) = f(y)$,
- for all $x, y \in G$, if $x^{-1}y \in H$, then $f(x) = f(y)$.

Moreover, the following are equivalent:

- $f$ is constant on right cosets of $H$,
- for all $x, y \in G$, if $yx^{-1} \in H$, then $f(x) = f(y)$,
- for all $x, y \in G$, if $xy^{-1} \in H$, then $f(x) = f(y)$.

*Because:* this follows from 4.121 and the definitions of constant on left and right cosets.

### 4.4.2

Given any group $G$, set $X$ and function $f : U(G) \to X$, there is a subgroup $H$ of $G$ with the property that $f$ is constant on both the left and right cosets of $H$.

*Because:* cosets of the identity subgroup are singleton sets.

### 4.4.3

Let $H$ be a subgroup of a group $G$ and let $X$ be a set. A function $f : U(G) \to X$ that is constant on left (right) cosets of $H$ need not be constant on the right (left) cosets of $H$. Consider, for example, the symmetric group $S_3$ and the subgroup $H = \{1, a\}$ discussed in 4.122. Let $X = \{0, 1, 2\}$ and define $f : U(G) \to X$ as follows:

<table>
<thead>
<tr>
<th>$x$</th>
<th>1</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f(x)$</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

Then $f$ is constants on the left cosets $H$, $bH = eH = \{b, e\}$ and $cH = dH = \{c, d\}$ but it is not constant on the right cosets $H$, $bH = dH = \{b, d\}$ and $cH = eH = \{c, e\}$. 

---

*Exercise:* Consider the function $f : U(G) \to X$ as defined in 4.4.3. Suppose $H$ is a subgroup of $G$. Prove that if $f$ is constant on left cosets of $H$, then $f$ is also constant on right cosets of $H$. 

---

*Exercise:* Let $H$ be a subgroup of a group $G$ and let $X$ be a set. Define a function $f : U(G) \to X$ by $f(x) = xH$. Prove that $f$ is constant on cosets of $H$.
4.4. THE HIDDEN SUBGROUP PROBLEM

4.4.14 Let $X$ be a set and let $H_1 < H_2 < \cdots$ be an ascending chain of subgroups of a group $G$ and let $f : U(G) \to X$ be a function that is constant on the left cosets of each $H_i$. Then there is a subgroup $H$ of $G$ for which $H_i < H$ and $f$ is constant on the left cosets of $H$.

Similarly, if $f$ is constant on the right cosets of each $H_i$, then there is a subgroup $H$ of $G$ for which $H_i < H$ and $f$ is constant on the right cosets of $H$.

Because: $H = \bigcup H_i$ is a subgroup of $G$ that satisfies the conclusion. To see that $H$ is a subgroup, note that the $G$-identity element $e \in H_1 \subset H$. If $x, y \in H$, then $x \in H_i$ and $y \in H_j$ for some $i$ and $j$. Let $k = \max\{i, j\}$. Then $x, y, x * y \in H_k \subset H$. To see that $f$ is constant on the left cosets of $H$, suppose $y^{-1} x \in H$. Then $y^{-1} x \in H_i$ for some $i$. This implies that $f(y) = f(x)$ since $f$ is constant on the left cosets of $H_i$.

The proof for right cosets is similar.

4.4.15 Assume that $G$ is a group, $X$ is a set and $f : U(G) \to X$ is a function. Then there is a maximal subgroup $H$ of $G$ for which $f$ is constant on the left cosets of $H$. Similarly, there is a maximal subgroup $H'$ of $G$ for which $f$ is constant on the right cosets of $H'$.

Because: the collection $\mathcal{H}$ of subgroups of $G$ having the property that $f$ is constant on left cosets is partially-ordered by inclusion, is nonempty by 4.412 and contains an upper bound for each ascending chain by 4.414. By Zorn’s Lemma [48], $\mathcal{H}$ contains a maximal element. The proof for right cosets is analogous.

4.42 Statement of Problem. Many quantum algorithms that provide an exponential speedup over classical algorithms are particular instances of solutions to The Hidden Subgroup Problem. We give a slight reformulation of this problem below. This change in perspective is motivated by the conclusions in 4.412 and 4.415.

Let $G$ be a group, let $X$ be a set and let $f : U(G) \to X$ be a function. Find a maximal subgroup $H$ of $G$ with the property that $f$ is constant on the left cosets of $H$.

The problem for right cosets is analogous. The left and right coset problems coincide for abelian groups. When working with finitely-presented groups, finding a subgroup usually means computing a presentation for it.

4.421 In general, the solution to the Hidden Subgroup Problem for left cosets is not equal to the solution to the problem for right cosets. Consider, for example, the group $G = S_3$ and the function $f$ described in 4.413. The subgroup $H = \{1, a\}$ solves the Hidden Subgroup Problem for left cosets while $\{1\}$ solves the problem for right cosets.

4.422 The adjunction between the free group and underlying set functors involves a bijection between functions $X \to U(G)$ and homomorphisms $F(X) \to G$ (see 4.142). The definition of the Hidden Subgroup Problem in terms of functions $U(G) \to X$ suggests that hidden subgroups might involve a right adjoint to $U$. However, we have the following.
The underlying set functor $U : \text{Set} \to \text{Group}$ does not have a right adjoint.

Because: the trivial group $\{e\}$ is an initial object in $\text{Group}$, however, its underlying set is not the empty set. Hence, $U$ does not preserve all colimits. This contradicts 2.768.

### 4.43 The Hidden Subgroup Problem as a Kan Extension

Despite the conclusion of 4.422, we have the following.

The Hidden Subgroup Problem can be formulated as a Kan extension.

Because: Let $G$ be a group, $S$ be a set, and $f : G \to S$ be a function. If $H$ is a subgroup of $G$, then we say that $f$ is constant on the cosets of $H$ if the following holds: for any $x, y \in G$, if $x^{-1} * y \in H$, then $f(x) = f(y)$ (where $*$ is the multiplication operation of $G$). Note that every $f$ is constant on the cosets of the trivial subgroup $\{e\}$ (where $e$ is the multiplicative identity element of $G$). The Hidden Subgroup Problem is to find a maximal subgroup $H_\ast$ which has the property that $f$ is constant on its cosets. The challenge is to calculate and present $H_\ast$ in an effective manner (e.g., by finding a set of generators).

The Hidden Subgroup Problem can be solved by finding the left adjoint of a forgetful functor. This is equivalent to computing a Kan extension. Below we describe this forgetful functor and its adjoint.

Fix a group $G$ and a set $S$. Consider the following category $\mathcal{C}_{G,S}$. The objects of $\mathcal{C}_{G,S}$ are triples $(f, H, f')$ where $H$ is a subgroup of $G$ and $f : G \to S$ and $f' : G/H \to S$ are functions satisfying $f = f' \circ \pi_H$ (with $\pi_H : G \to G/H$ the natural map $g \mapsto gH$ induced by $H$) as shown in the diagram below. Commutativity of this diagram is equivalent to the condition that $f$ is constant on the cosets of $H$.

\[
\begin{array}{ccc}
G & \xrightarrow{f} & S \\
\downarrow{\pi_H} & & \downarrow{f'} \\
G/H & \xrightarrow{\pi_1} & G/H_2 \\
\end{array}
\]

In $\mathcal{C}_{G,S}$, the set of morphisms from an object $(f_1, H_1, f'_1)$ to an object $(f_2, H_2, f'_2)$ is defined by

\[
\mathcal{C}_{G,S}((f_1, H_1, f'_1), (f_2, H_2, f'_2)) = \begin{cases} 
\{(H_2, H_1)\} & \text{if } f_1 = f_2 \text{ and } H_2 \text{ is a subgroup of } H_1 \\
\phi & \text{otherwise.} 
\end{cases}
\]  

(4.1)

In words, there is a unique morphism from $(f_1, H_1, f'_1)$ to $(f_2, H_2, f'_2)$ if and only if $f_1 = f_2$ and $H_2$ is a subgroup of $H_1$. This is illustrated by the commutative diagram below where the induced map $i : G/H_2 \to G/H_1$ defined by $i(gH_2) = gH_1$ is surjective.

\[
\begin{array}{ccc}
G & \xrightarrow{f} & S \\
\downarrow{\pi_1} & & \downarrow{f_2} \\
G/H_1 & \xrightarrow{i} & G/H_2 \\
\end{array}
\]

In $\mathcal{C}_{G,S}$, composition of morphisms $(f, H_1, f'_1) \circ (H_2, H_1) \circ (f, H_2, f'_2) \circ (H_3, H_2) \circ (f, H_3, f'_3)$ is $(H_3, H_1)$.
The other category used to characterize the Hidden Subgroup Problem as a Kan extension is constructed as follows. Consider the collection of functions from the group $G$ to the set $S$. Let $\mathcal{D}_{G,S}$ be the discrete category generated by that set. That is, $\mathcal{D}_{G,S}$ has functions $G \to S$ as objects and, for any two objects $f_1$ and $f_2$ of $\mathcal{D}_{G,S}$, the set of morphisms is defined by

$$\mathcal{D}_{G,S}(f_1, f_2) = \begin{cases} \{ \text{id}_f \} & \text{if } f_1 = f_2 = f \\ \varnothing & \text{otherwise.} \end{cases} \quad (4.2)$$

There is a forgetful functor $U$ from $\mathcal{C}_{G,S}$ to $\mathcal{D}_{G,S}$ defined by $U(f, H, f') = f$ on objects and

$$f((f, H_1, f_1') \xrightarrow{H_2, H_3} (f, H_2, f_2')) = \text{id}_f$$
on morphisms. More interesting than this forgetful functor is its left adjoint, $L$, which solves the Hidden Subgroup Problem\(^1\).

![Diagram](https://via.placeholder.com/150)

Let $f : G \to S$. Then $L(f) = (f, H_*, f')$ where $H_*$ is the largest subgroup of $G$ that satisfies the property. In other words, there may be many subgroups $H$ of $G$ such that $f$ is constant on the cosets $G/H$ but $L$ finds a maximal one.

The universal property of the adjoint functor is the following: for any object $f : G \to S$ of $\mathcal{D}_{G,S}$ and any object $(g, H, g')$ of $\mathcal{C}_{G,S}$, there is a bijection between sets of morphisms:

$$\mathcal{C}_{G,S}(L(f), (g, H, g')) \cong \mathcal{D}_{G,S}(f, g). \quad (4.3)$$

If $f \neq g$, then both sides of the equation are empty. If $f = g$, then the right-hand side is non-empty (in fact, it has a unique element), hence, the left-hand side must be non-empty. This implies that $H$ is a subgroup of $H_*$ (where $L(f) = (f, H_*, f')$). That is, $f$ is constant on the cosets of $H_*$ and if $H_1$ is any other such subgroup, then $H_1$ is a subgroup of $H_*$. 

In our discussion so far, we have assumed that the maximal subgroup $H_*$ required in the definition of the adjoint $L$ exists. To prove the existence of $H_*$, let $\mathcal{A}$ be the collection of subgroups $H$ of $G$ having the property that $f$ is constant on the cosets of $H$. $\mathcal{A}$ is non-empty since $\{e\} \in \mathcal{A}$. $\mathcal{A}$ is partially-ordered by inclusion $\subseteq$. Note that if $H_1 \subseteq H_2 \subseteq \cdots$ is a chain in $\mathcal{A}$, then $H = \bigcup_{k=1}^{\infty} H_k$ is in $\mathcal{A}$ and is an upper bound on the set of subgroups $H_k$. It is, therefore, a consequence of Zorn’s Lemma that $\mathcal{A}$ has a maximal element, $H_*$. 

How is $L(f)$ constructed? On page 234, Section X.1 of [?], the usual formula for constructing a left adjoint is given. In order to understand that construction in our context, note that for $f : G \to S$ in $\mathcal{D}_{G,S}$, $U : \mathcal{C}_{G,S} \to \mathcal{D}_{G,S}$ induces a category $(f \downarrow U)$ whose objects are maps $f \to U(g)$ in $\mathcal{D}_{G,S}$ and whose morphisms are commuting triangles. There is a forgetful functor $Q : (f \downarrow U) \to \mathcal{C}_{G,S}$ which takes $f \to U(g)$ to $g$. With this in mind, the formula for constructing the left adjoint is

$$L(f) = \text{colim}(Q : (f \downarrow U) \to \mathcal{C}_{G,S}). \quad (4.4)$$

\(^1\)This result is consistent with the slogan *Important concepts occur as adjoints to trivial functors.* which is due to F. W. Lawvere and is pursued in [132].
This is the origin of 2.824 on which the classical algorithm is based. Finally, we note that the left adjoint \( L \) can be described as a Kan extension of the identity functor \( \text{id} : C_{G,S} \to C_{G,S} \) along the forgetful functor \( U : C_{G,S} \to D_{G,S} \).

\[
\begin{array}{ccc}
D_{G,S} & \overset{L}{\longrightarrow} & C_{G,S} \\
\downarrow U \quad & & \quad \downarrow \text{id} \\
C_{G,S} & \quad \quad & \quad \quad C_{G,S}
\end{array}
\]

The formula for this Kan extension given in [102], Page 238, Section X.3 is

\[
L(f) = \lim_{\rightarrow} (\text{id} \circ Q : (f \downarrow U) \to C_{G,S} \to C_{G,S}).
\]

which reduces to Equation 4.4 above. Although we are in fact looking for a left adjoint, we might want to keep the language of Kan extensions due to the availability of a computational algorithm and since Kan liftings and homotopy Kan extensions and liftings are the right way to generalize.

### 4.44 Examples

In the following subsections we review several examples of hidden subgroup problems.

#### 4.441 Deutsch’s Problem

Let \( G = \mathbb{Z}_2 \) and let \( X = \{0, 1\} \). The four possibilities for \( f : U(G) \to X \) are tabulated below.

<table>
<thead>
<tr>
<th>( x )</th>
<th>( f_0(x) )</th>
<th>( f_1(x) )</th>
<th>( \text{id} )</th>
<th>( \zeta )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

\( f_0 \) and \( f_1 \) are constant. The other two functions are balanced. \( \mathbb{Z}_2 \) has two subgroups: \( \{0\} \) and itself. Solutions to the Hidden Subgroup Problem for the four possible choices of \( f \) are \( H_0 = H_1 = \mathbb{Z}_2 \) and \( H_{\text{id}} = H_{\zeta} = \{0\} \). The problem is to determine whether a given function \( f \) is constant or balanced. This clearly can be determined by evaluating the function on both inputs. Deutsch’s algorithm makes the determination using only one function evaluation.

In this instance of the Hidden Subgroup Problem, \( G = \mathbb{Z}_2 \), \( x = \{0, 1\} \), and \( f \) is the given function. \( G \) only has two subgroups \( H = \{e\} \) and \( H = \mathbb{Z}_2 \).

\[
\begin{array}{ccc}
\mathbb{Z}_2 & \overset{f}{\longrightarrow} & \{0, 1\} \\
\downarrow \pi \quad & & \quad \downarrow f' \\
G/H & \quad \quad & \quad \quad G/H
\end{array}
\]

If \( H = \{e\} \) then \( f = f' \) is balanced. Otherwise, \( H = \mathbb{Z}_2 \) and \( f \) is constant.

#### 4.442 Deutsch-Jozsa

This is a generalization of Deutsch’s Algorithm to more complicated inputs. Let \( \mathbb{Z}_2^n = \mathbb{Z}_2 \times \mathbb{Z}_2 \times \cdots \times \mathbb{Z}_2 \) to be thought of as strings of \( n \) zeros and ones. Addition in the group is point-wise. Given a function \( f \) from the group \( \mathbb{Z}_2^n \) to the set \( \{0, 1\} \) that is either constant or balanced, we are asked to determine which case holds.
In this instance of the Hidden Subgroup Problem, \( G = \mathbb{Z}_2^n \) and \( S = \{0, 1\} \).

If \( H = \mathbb{Z}_2^n \), then \( f \) is constant. Otherwise we are assured that \( f \) is balanced.

### 4.4.3 Simon’s Problem

Let \( G = \mathbb{Z}_2^n \) and let \( X \) be any finite set. Let \( G = \mathbb{Z}_2^n \). Let \( S \) also equal the set \( \mathbb{Z}_2^n \). We are given a map \( f : \mathbb{Z}_2^n \to \mathbb{Z}_2^n \) and assured that there exists \( c \in \mathbb{Z}_2^n \) such that for all \( x \in \mathbb{Z}_2^n \)

\[
f(x) = f(x + c).
\] (4.6)

We are asked to find \( c \). Note that \( H = \{0, c\} \) is a subgroup of \( \mathbb{Z}_2^n \), and \( \{x, x + c\} \) is the coset of this subgroup containing \( x \). Equation 4.6, therefore, asserts that \( f \) is constant on the cosets of \( H \).

### 4.4.4 Shor’s Order-Finding Problem

This is a generalization of Simon’s Periodicity Algorithm from the finite group \( \mathbb{Z}_2^n \) to the infinite group \( \mathbb{Z} \). Let \( G = \mathbb{Z} \). Let \( N \) be a number that is not prime that we seek to factor. Assume \( a < N \) is such that \( \text{GCD}(a, N) = 1 \). Consider the function

\[
f_{a,N}(x) = a^x \mod N.
\] (4.7)

The set of possible outputs of \( f_{a,N} \) is \( S = \{0, 1, 2, 3, \ldots, N-1\} \). For some \( r \), \( f_{a,N} \) has period \( r \) which implies

\[
f_{a,N}(r) = a^r \mod N = 1
\] (4.8)

and

\[
f_{a,N}(x) = f_{a,N}(x + r).
\] (4.9)

We are asked to find \( r \). Note that \( r \in \mathbb{Z} \) generates the subgroup \( H = \{0, \pm r, \pm 2r, \pm 3r, \ldots\} \).

Once we have found \( r \), we compute the factors of \( N \) classically.
4.445 **Abelian Stabilizer Problem.** Now that we have a characterization of the Hidden Subgroup Problem as a Kan extension, we revisit the Abelian Stabilizer Problem that we first discussed in 4.3. Let $G$ be an abelian group and let $S$ be a set with a group action $f : G \times S \to S$. By a group action we mean that $f$ satisfies the following for all $g, h \in G$ and all $s \in S$:

$$f(gh, s) = f(g, f(h, s))$$

and

$$f(e, s) = s.$$  

(4.10)

(4.11)

We want to find those elements of $G$ that stabilize all the elements of $S$:

$$H = \{ g \in G \mid \forall s \in S, f(g, s) = s \}.$$  

(4.12)

The stabilizer $H$ is a subgroup of $G$ and the group action $f$ is constant on its cosets.

### 4.446 Other Examples

Other instances of the Hidden Subgroup Problem such as the Discrete Logarithm and Period-Finding Problems (see page 241 of [112]) formulated as Kan extensions in a manner similar to the cases detailed above.

### 4.45 Quantum Algorithms in Group Theory

[18] gives a survey of quantum computing basics and of standard algorithms including those discovered by Deutsch-Jozsa, Shor and Grover. The author also surveys Watrous’ algorithms for solvable groups.

[130] gives an algorithm for computing a particular class of left Kan extensions. Let $G$ be a finite solvable group, viewed as a category. See also [129]. Let $\{e\}$ be the trivial subgroup, also construed as a category, and let $X : 1 \to \text{Set}$ be the functor that maps the object of 1 to a singleton set. A left Kan extension of $X$ along the inclusion functor $1 \to G$ computes the set of elements of $G$. The algorithm of [130] computes the order of this set. The group is given as a finite black box. Multiplication and inversion are assumed to be unit cost operations performed by an oracle.

Recall that a **normal series** for a group is a sequence of subgroups

$$G = G_0 \geq G_1 \geq \cdots \geq G_n = 1$$

in which $G_{i+1} \triangleleft G_i$ for all $i$. The **factor groups** for the normal series are the groups $G_i/G_{i+1}$ for $i = 0, 1, \ldots, n-1$. A finite group is **solvable** if it has a normal series whose factor groups are cyclic of prime order. In particular, every abelian group is solvable. The permutation group $S_5$ is an example of a non-solvable group [116].
Chapter 5

Abelian Categories

The quantum Fourier transform is a key factor in quantum factoring and many other quantum algorithms that are known to be exponentially faster than their classical counterparts. Fourier analysis, more generally, is the study of objects defined on topological groups [91]. In this chapter we explore the possibility of adapting the quantum Fourier transforms to a more general context in which a group structure is available: abelian categories.

5.1 Zero Objects and Kernels

In a category with a zero object, there is a unique, trivial morphism between any two objects. A zero object also gives ways to identify special morphisms in a category.

5.1.1 Zero Objects and Morphisms. An object in a category is a zero object if it is both initial (see 2.62) and terminal (see 2.52).

Not every category has a zero object. In Set, for example, the empty set $\phi$ and a singleton set $1$ are not isomorphic, hence, no object can be both initial and terminal in Set. In the category of abelian groups, any one element group is a zero object. In the category of modules over any ring, the zero module is a zero object. Similarly, the category $\text{Ban}_{\infty}$ of Banach spaces and bounded linear mappings has the trivial (zero dimensional) Banach space as zero object. In the category of pointed sets (see [102]), any one point set is a zero object.

5.1.11 If $0$ and $0'$ are zero objects in a category $A$, then $A(0, 0')$ consists of a single morphism and this is an isomorphism.

Because: $0 \to 0' \to 0$ and $0' \to 0 \to 0'$ are isomorphisms since $A(0, 0) = \{id_0\}$ and $A(0', 0') = \{id_{0'}\}$.

5.1.12 Let $A$ admit a zero object. A morphism $f \in A(A, B)$ is a zero morphism if for some zero object $0 \in \text{ob}(A)$, $f = i_B \circ !_A$.

\[
\begin{array}{c}
A \xrightarrow{f} B \\
i_A \downarrow \quad \downarrow i_B \\
0
\end{array}
\]
That is, \( f \) factors through some zero object. We may show that in this case, \( f \) factors through any zero object of \( \mathcal{A} \). That is, being a zero morphism is in fact a property of the morphism alone. It is independent of the choice of zero object. The next result shows that zero morphisms do exist in a category with a zero object. In fact, there is precisely one for each ordered pair of objects. This implies that no hom set is empty: each contains at least the zero morphism.

5.113 Let \( \mathcal{A} \) admit a zero object. A morphism \( f \in \mathcal{A}(A, B) \) is a zero morphism if and only if given any zero object, \( 0 \), \( f = \text{id}_B \circ !_A \).

Because: \( f = \text{id}_B \circ !_B = (\text{id}_B \circ \varphi^{-1}) \circ (\varphi \circ !_A) \) where \( \varphi : 0' \to 0 \) is an isomorphism of zero objects.

5.114 Let \( \mathcal{A} \) be a category with a zero object. Then \( \forall A, B \in \text{ob}(\mathcal{A}) \), there is a unique zero morphism \( 0_{A,B} \in \mathcal{A}(A, B) \). Moreover, for any zero object \( 0 \), \( 0_{A,B} = !_A \circ !_B \).

Because: existence follows from the formula for \( 0_{A,B} \) while 5.113 establishes uniqueness.

5.115 Let \( \mathcal{A} \) admit a zero object, \( 0 \). For any object \( A \) of \( \mathcal{A} \), \( 0_{0,A} = !_A \). Similarly, for any object \( B \) of \( \mathcal{A} \), \( 0_{B,0} = !_B \).

Because: \( !_0 = \text{id}_0 \).

5.116 Let \( \mathcal{A} \) admit a zero object. If \( f \in \mathcal{A}(B, C) \) is a zero morphism and \( g \in \mathcal{A}(A, B) \) then \( f \circ g \) is a zero morphism.

Because: \( f \circ g = \text{id}_C \circ !_B \circ g = \text{id}_C \circ !_A = 0_{A,C} \).

5.117 Let \( \mathcal{A} \) admit a zero object. If \( f \in \mathcal{A}(A, B) \) is a zero morphism and \( g \in \mathcal{A}(B, C) \) then \( g \circ f \) is a zero morphism.

Because: \( g \circ f = g \circ \text{id}_B \circ !_A = \text{id}_C \circ !_A = 0_{A,C} \).

5.118 Let \( \mathcal{A} \) admit a zero object. If \( g \circ f \) is a zero morphism and \( g \) is monic then \( f \) is a zero morphism.

Because: \( g \circ f = \text{id}_C \circ !_A = (g \circ \text{id}_B) \circ !_A = g \circ \text{id}_B = g \circ 0_{A,B} \). g monic implies \( f = 0_{A,B} \).

5.119 Let 0 be a zero object in a category \( \mathcal{A} \). For any object \( A \) of \( \mathcal{A} \), \( i_A \in \mathcal{A}(0, A) \) is a monomorphism. Similarly, \( !_A \in \mathcal{A}(A, 0) \) is an epimorphism.

Because: \( i_A \circ x = i_A \circ y \) implies that \( x \) and \( y \) have common domain and have 0 as codomain. Since 0 is terminal, \( x = y \). This implies that \( i_A \) is a monomorphism. The epimorphism proof is similar.
5.12 **Kernels.** A kernel of a morphism \( f \in \mathcal{A}(A, B) \) in a category \( \mathcal{A} \) which admits a zero object is an equalizer of \( f \) and \( 0_{A,B} \).

\[
\begin{array}{ccc}
\text{ker}(f) & \xrightarrow{i} & A \\
& \xrightarrow{f} & 0_{A,B} \\
& \xrightarrow{0} & B.
\end{array}
\]

5.121 **A kernel induces a monomorphism, hence, a subobject.**  
*Because:* equalizers are monomorphisms. See 2.545.

5.122 **A monomorphism in a category with a zero object need not be a kernel.**  
*Because:* See example 1.1.9.a in Volume II of [25].

5.123 **Let \( A \) admit a zero object.** For objects \( A \) and \( B \) of \( \mathcal{A} \), \((A, \text{id}_A)\) is a kernel of \( 0_{A,B} \).  
*Because:* see 1.1.8 in Volume II of [25].

5.124 **Let \( A \) admit a zero object and let \( f \in \mathcal{A}(A, B) \).** Among the conditions:

1. \( f \) is a monomorphism;
2. \((0, i_A)\) is a kernel of \( f \);
3. \( \forall C \in \text{ob}(\mathcal{A}), \forall g \in \mathcal{A}(C, A), f \circ g = 0_{C,B} \) implies \( g = 0_{C,A} \);

the following hold: \( 1) \Rightarrow 2), 1) \Rightarrow 3) \) and \( 2) \Leftrightarrow 3). \)

In particular, any monomorphism has a kernel. In an abelian category, each monomorphism is a kernel of some arrow. It is also true that in an abelian category every morphism has a kernel.  
*Because:* 1) \( \Rightarrow 2): f \circ i_A = i_B = 0_{A,B} \circ i_A \) and if \( f \circ g = 0_{A,B} \circ g \) for some \( g : X \to A \), then \( g = 0_{X,A} \) by 5.118. For 1) \( \Rightarrow 2), see 1.1.7 in Volume II of [25].

5.13 **Cokernels.** A cokernel of a morphism \( f \in \mathcal{A}(A, B) \) in a zero object category \((\mathcal{A}, 0)\) is an coequalizer of \( f \) and \( 0_{A,B} \).

\[
\begin{array}{ccc}
A & \xrightarrow{f} & B \\
& \xrightarrow{0_{A,B}} & \text{coker}(f).
\end{array}
\]

5.131 **Let \( A \) admit a zero object and let \( f \in \mathcal{A}(A, B) \).** The following are equivalent:

1. \( f \) is an isomorphism;
2. for any object \( C \) of \( \mathcal{A} \), \((A, f)\) is a kernel of \( 0_{B,C} \);
3. for any object \( C \) of \( \mathcal{A} \), \((B, f)\) is a cokernel of \( 0_{C,A} \).

Existence of a zero object, thus, gives us a way to identify special morphisms.  
*Because:* see 1.1.8 in Volume II of [25].
5.132 Let $A$ admit a zero object and let $f \in A(A, B)$. Among the following conditions,
1. $f$ is a epimorphism;
2. $(0, !_B)$ is a cokernel of $f$;
3. $\forall C \in \text{ob}(A), \forall g \in A(B, C), g \circ f = 0_{A,B}$ implies $g = 0_{B,C}$;
the following hold: $1) \Rightarrow 2)$ and $1) \Rightarrow 3)$ while $2) \iff 3)$.

In particular, any epimorphism has a cokernel. In an abelian category, each epimorphism is a cokernel of some arrow. It is also true that in an abelian category every morphism has a cokernel.

5.2 Additive Categories

5.21 Preadditive Categories. A (locally-small) category $C$ is preadditive if each set $C(A, B)$ is equipped with an abelian group structure in such a way that composition

$$C(A, B) \times C(B, C) \to C(A, C)$$

is a homomorphism.

5.211 In a preadditive category $C$, the following are equivalent.
1. $C$ has an initial object;
2. $C$ has a terminal object;
3. $C$ has a zero object.

In each case, a zero morphism $0_{A,B}$ coincides with the identity element of $C(A, B)$.

Because: see 1.2.3 in Volume II of [25].

5.212 Given two objects $A$ and $B$ in a preadditive category $C$, the following are equivalent:
1. $C$ admits a product $(P, p_A, p_B)$ of $A$ and $B$,
2. $C$ admits a coproduct $(P, s_A, s_B)$ of $A$ and $B$,
3. there exists an object $P$ and morphisms $p_A : P \to A$, $p_B : P \to B$, $s_A : A \to P$ and $s_B : B \to P$ with the properties

$$p_A \circ s_A = \text{id}_A, \quad p_B \circ s_B = \text{id}_B, \quad p_A \circ s_B = 0_{B,A}, \quad p_B \circ s_A = 0_{A,B}, \quad s_A \circ p_A + s_B \circ p_B = \text{id}_P.$$

Moreover, under these conditions, $s_A$ is the kernel of $p_B$, $s_B$ is the kernel of $p_A$, $p_A$ is the cokernel of $s_B$ and $p_B$ is the cokernel of $s_A$.

Because: see 1.2.4 in Volume II of [25].

5.213 A biproduct of objects $A$ and $B$ of a preadditive category $C$ is a structure $(P, p_A, p_B, s_A, s_B)$ satisfying condition 3) of 5.212.
### 5.3. ABELIAN CATEGORIES

**5.214** Given \( f : A \to B \) and \( g : A \to B \) in a preadditive category, the following are equivalent:

1. the equalizer of \( f \) and \( g \) exists,
2. the kernel of \( f - g \) exists,
3. the kernel of \( g - f \) exists.

*If any of the conditions holds then the three objects are isomorphic.*

*Because: see 1.2.8 in Volume II of [25].*

---

**5.22** **Additive Categories.** A category \( C \) is **additive** if it is preadditive and admits a zero object and biproducts of pairs of objects.

**5.221** \( C \) is additive iff \( C^{op} \) is.

*Because: each component of the definition of an additive category is its own dual.*

---

**5.3** **Abelian Categories**

A category \( A \) is **abelian** if and only if:

1. \( A \) admits a zero object;
2. \( A \) admits binary products and coproducts;
3. in \( A \) each monomorphism is a kernel and each epimorphism is a cokernel;
4. a kernel and a cokernel exists for each morphism of \( A \).

By condition 2), each monomorphism is an equalizer and each epimorphism is a coequalizer.

The category \( \text{Ab} \) of abelian groups is an abelian category as is the category of right (respectively, left) modules over any ring \( R \). We will see that an abelian category admits all finite limits and finite colimits. Moreover, each may be construed as an \( \text{Ab} \)-enriched category.

**5.302** A category \( A \) is an abelian if and only if \( A^{op} \) is.

*Because: each component of the definition is its own dual (e.g., the dual of a zero object is a zero object).*

---

**5.303** If \( J \) is a small category and \( A \) is an abelian category then \( A^J \) is abelian.

*Because: see 1.4.4 in Volume II of [25].*

---

**5.31** **Special Morphisms in Abelian Categories.** We have seen that a zero object gives a way to identify special morphisms. The conditions are strengthened if the category is abelian.

**5.311** Let \( A \) be abelian. For a morphism \( f \in \mathcal{A}(A, B) \), the following are equivalent:
1. \( f \) is a monomorphism;
2. \((0, i_A)\) is a kernel of \( f \),
3. \( \forall C \in \text{ob}(A), \forall g \in A(C, A), f \circ g = 0_{C,B} \) implies \( g = 0_{C,A} \).

Because: If \( A \) admits a zero object then 1) \( \Rightarrow \) 2), 1) \( \Rightarrow \) 3), and 2) \( \Leftrightarrow \) 3). To prove that 2) or 3) implies 1) requires that \( A \) be abelian, not simply that \( A \) admits a zero object. See 1.5.4 in Volume 2 of [25].

5.312 Let \( A \) be abelian. For a morphism \( f \in A(A, B) \), the following are equivalent:
1. \( f \) is an epimorphism;
2. \((0, !_B)\) is a cokernel of \( f \),
3. \( \forall C \in \text{ob}(A), \forall g \in A(B, C), g \circ f = 0_{A,B} \) implies \( g = 0_{B,C} \).

Because: this is the dual of 5.311.

5.313 Let \( A \) be an abelian category and let \( f \in A(A, B) \). The following are equivalent:
1. \( f \) is an isomorphism;
2. \( f \) is a monomorphism and an epimorphism.

Because: see 1.5.1 in Volume II of [25].

5.314 Let \( A \) be an abelian category, let \( f \in A(A, B) \) be a monomorphism, and let \((Q,q)\) be a cokernel of \( f \). Then \((A,f)\) is a kernel of \( q \):

\[
\begin{array}{ccc}
A & \overset{f}{\longrightarrow} & B \\
& \overset{q}{\longrightarrow} & Q \\
\end{array}
\]

5.315 Let \( A \) be an abelian category, let \( f \in A(A, B) \) be an epimorphism, and let \((K,k)\) be a kernel of \( f \). Then \((A,f)\) is a cokernel of \( k \):

\[
\begin{array}{ccc}
K & \overset{k}{\longrightarrow} & A \\
& \overset{f}{\longrightarrow} & B \\
\end{array}
\]

That is, each monomorphism in an abelian category is a kernel of its cokernel and each epimorphism is a cokernel of its kernel.

5.316 An abelian category is finitely complete and finitely cocomplete.

Because: see 1.5.3 in Volume II of [25].

5.317 Every morphism \( f \) in an abelian category can be factored uniquely up to isomorphism as \( f = i \circ p \), where \( i \) is a monomorphism and \( p \) is an epimorphism. Moreover, \( i \) is the kernel of the cokernel of \( f \) and \( p \) is the cokernel of the kernel of \( f \).

Because: see 1.5.5 in Volume 2 of [25].
5.32 Hidden Subgroup Problems in Abelian Categories. We explored formulating the Hidden Subgroup Problem in the general context of an abelian category and the potential of quantum computers to efficiently solve such problems.
Chapter 6

Classical Algorithms for Computing Kan Extensions

6.1 The Todd-Coxeter Coset Enumeration Procedure

Since coset enumeration is an example of a Kan extension, analysis of the Todd-Coxeter Procedure and later coset enumeration techniques provides information about the complexity of Kan extension calculations. See 4.2. It is known that if the index of the subgroup $H$ in $G$ is finite, then the Todd-Coxeter algorithm terminates in finite time $[103, 123]$ even if the ambient group $G$ is infinite. These proofs, however, do not provide a bound on the computational space or time that the algorithms may require even in the case that the finitely presented group turns out to be the trivial group. In fact, there is no computable bound, in terms of the length of the input and a hypothetical index, on the number of cosets which need to be defined to obtain a complete coset table $[77, 119]$.

6.2 The Carmody-Walters Algorithm

The Todd-Coxeter Procedure described in 4.21 is a special case of the Carmody-Walters algorithm for computing left Kan extensions. Given finite sets of objects and generating arrows of a category $\mathcal{A}$, a finite presentations of a category $\mathcal{B}$, a functor $F : \mathcal{A} \rightarrow \mathcal{B}$ and an action $X : \mathcal{A} \rightarrow \text{Set}_f$ with each $X(A)$ finite, the Carmody-Walters algorithm finds a left Kan extension $(L, \epsilon)$ of $X$ along $F$ if each $L(B)$ is finite. It consists of an action $L : \mathcal{B} \rightarrow \text{Set}_f$ and a natural transformation $\epsilon : X \Rightarrow L \circ F$ satisfying the universal mapping property described in 2.82.

The Kan extensions algorithm involves constructing four types of tables. There is one $\epsilon$-table for each $A \in |\mathcal{A}|$, one $L$-table for each $B \in |\mathcal{B}|$, one relation table for each relation of $\mathcal{B}$ and one naturality table for each $A \in |\mathcal{A}|$. In the Todd-Coxeter procedure, the $\epsilon$-table is trivial. This is not the case for general Kan extensions. Each $\epsilon$-table that occurs in the Carmody-Walters has two columns and a fixed number of rows. The $L$- and relation-tables also have a fixed number of columns (although different tables may have different numbers of columns) and an unknown number of rows. Initially, the entries of all tables except the $\epsilon$-table are blank (or undefined). As the algorithm proceeds, we successively add entries 0, 1, 2, ... into the $L$-tables, fill in consequences of those new entries in the relation and naturality tables then deal with coincidences that may be imposed by those tables.

For each $A \in |\mathcal{A}|$ there is an $\epsilon$-table constructed as follows.
where $X(A) = \{0, \ldots, n_{X(A)} - 1\}$. For each $B \in |\mathcal{B}|$, there is an $L$-table constructed as follows. It has one column for $B$ and an additional column for each generator $g \in \mathcal{B}(B, B')$ of $\mathcal{B}$ having domain $B$.

\[
\begin{array}{c|c|c}
L(B) & L(B') & \cdots \\
\end{array}
\]

We leave the bottom of each column open since the number of rows (i.e., the size of the set $L(B)$) is not known a priori.

For each relation $r$ of $\mathcal{B}$, we construct a relation table as follows. Assume that the relation is $b_m \circ \cdots \circ b_1 = b'_n \circ \cdots \circ b'_1$ and that it has domain $B$ and codomain $B'$.

\[
\begin{array}{c|c|c|c|c}
L(b_1) & L(b_m) & L(b'_1) & L(b'_n) \\
L(B) & \cdots & L(B') & L(B) & \cdots & L(B') \\
\end{array}
\]

As in the $L$-table, we leave the bottom of each column of the relation tables open since the number of rows (i.e., the size of the set $L(B)$) is not known a priori. In the special case of a relation $x_1 \cdots x_m = 1$, we construct the relation table as follows.

\[
\begin{array}{c|c|c|c}
L(x_1) & L(x_m) \\
L(*) & \cdots & L(B) & L(B) \\
\end{array}
\]

The pseudo-code shown in Listing 2 sketches the algorithm. The manner in which one fills in the details of this sketch leads to distinct implementations and can have a dramatic effect on the performance of the algorithm in some cases. In line 3, for example, we may (1) define a single new entry in an $L$-table, or (2) define enough entries to complete a row of an $L$-table or (3) make some other choice.

```
1 initialize tables
2 while (there are undefined entries in the $\epsilon$-tables or the $L$-tables) {
3   define new entries in these tables
4   fill in consequences in the relation tables
5   while (there are coincidences in the relation and naturality tables) {
6     deal with coincidences that arise in the relation and naturality tables
7     fill in consequences in all the tables
8   }
9}
```

Listing 2: Pseudo-code for the Carmody-Walters algorithm

Figure 6.1 illustrates the evolution of the algorithm. As shown in 2.824 and 2.661, left Kan extensions can be constructed from two particular kinds: coproducts and coequalizers. For each object $B \in |\mathcal{B}|$, we
must construct a finite set $L(B)$. The top of the figure gives a formula for this set. It is a disjoint union on which an equivalence relation is imposed. Components $B(F(A), B) \times X(A)$ of the disjoint union are indexed by objects $A \in |\mathcal{A}|$.

In the figure, we draw objects of $\mathcal{A}$ along a horizontal line and use these points along the line as indices for the components. The vertical extent of each component is $X(A)$ and is known from the input data. the horizontal extent is $B(F(A), B)$ and is not known a priori, hence, is drawn using a dotted line. In the Todd-Coxeter procedure, for example, we saw in Table 4.1 on page 70 that the number of cosets computed during execution of the algorithm can differ considerably from the actual number of cosets. A similar phenomenon occurs in general when using the Carmody-Walters Algorithm.

Items in blue are related to the equivalence relation in the definition of $L(B)$. The equivalence relation on the disjoint union is generated by all pairs of points such as those connected by a blue arc.

Items in green are related to the definition of $L$ on morphisms.

$$L(B) = \left( \sum_{A \in \mathcal{A}} B(F(A), B) \times X(A) \right) \sim L(\beta)[b, x] = [\beta \circ b, x]$$

![Figure 6.1: Graphical interpretation of the Carmody-Walters algorithm](image)

6.3 Examples

In the following subsections, we describe a variety of special cases of the general left and right Kan extension construction. We discuss classical algorithms for computing these special cases in $\mathbf{Set}$. 
6.31  **Products.** A product of sets $X$ and $Y$ is a triple $(X \times Y, \pi_0, \pi_1)$ with $X \times Y \xrightarrow{\pi_0} X$ and $X \times Y \xrightarrow{\pi_1} Y$ and which enjoys the property that if $(Z, u, v)$ is any such triple, then there is a unique $Z \xrightarrow{(u,v)} X \times Y$ for which the following commutes.

\[
\begin{array}{c}
X & \xleftarrow{\pi_0} & X \times Y & \xrightarrow{\pi_1} & Y \\
\downarrow{u} & & \downarrow{(u,v)} & & \downarrow{v} \\
Z & & & & \\
\end{array}
\]

This is the universal mapping property defined in 2.53.

A product is a particular kind of limit and, by 2.816, limits may be computed as right Kan extensions. Moreover, all right Kan extensions may be computed from two particular kinds of limits: products and equalizers. As discussed above, this follows from 2.813 and 2.564. If we unwrap the layers in these theorems, we find that products are expressed as the following right Kan extension.

\[
\begin{array}{c}
\{0,1\} \xrightarrow{!} \{\ast\} \\
\downarrow{(\pi_0, \pi_1)} & \downarrow{X \times Y} \\
(X,Y) & \xrightarrow{\text{Set}} & X \times Y \\
\end{array}
\]

We know from 3.12 that $\text{Set}$ has all small products, hence, it has binary products. The cartesian product gives one solution to this universal mapping property.

By 3.2 we know that the category of finite sets is equivalent to the category $\Delta$ of simplicial sets, hence, in working with a finite set $X$ with $|X| = n$, we may instead work with its representative $\{0, \ldots, n - 1\}$ in $\Delta$. From this we obtain a simple formula for categorical products of finite sets: If $X = \{0, \ldots, m - 1\}$ and $Y = \{0, \ldots, n - 1\}$, then

\[
X \times Y = \{0, \ldots, (m \times n) - 1\} \quad \pi_0(k) = \lfloor m/n \rfloor \quad \text{and} \quad \pi_1(k) = k \mod n
\]

6.311  [50] presents classical circuits for multiplying binary integers and analyzes the time and space resources that they require. For two $n$-bit integers, array multipliers operate in $\Theta(n)$ time and have $\Theta(n^2)$ size. An array multiplier circuit integrates carry-save addition circuits to sum the partial products. Wallace-tree multipliers require $\Theta(\lg(n))$ time and are also of size $\Theta(n^2)$. A Wallace-tree multiplier integrates both carry-save and carry-lookahead addition circuits.

6.32  **Equalizers.** Given morphisms $f, g : X \to Y$ in a category $\mathcal{C}$, an equalizer of $f$ and $g$ is a pair $(E, e)$ with $E$ an object and $e : E \to X$ a morphism for which $f \circ e = g \circ e$. Given any such pair $(E', e')$, there is a unique $\varphi : E' \to E$ for which $e' = e \circ \varphi$ as shown below.

\[
\begin{array}{c}
E' \xrightarrow{e'} E \\
\downarrow{\varphi} & \downarrow{e} \\
X & \xrightarrow{g} Y \\
\end{array}
\]

This is the universal mapping property defined in 2.54. If $\mathcal{C} = \text{Set}$, then

\[
E = \{x \in X \mid f(x) = g(x)\}
\]
and \( e : E \to X \) is the inclusion function as shown in 3.121. The subset \( E \) of \( X \) is also determined by the characteristic function \( \chi : X \to \{0, 1\} \) where

\[
\chi(x) = \begin{cases} 
1 & \text{if } f(x) = g(x) \\
0 & \text{otherwise.}
\end{cases}
\]

6.321 The classical complexity of calculating an equalizer is \( O(n) \). We evaluate the characteristic function \( \chi \) defined in 6.32 once for each \( x \in X \).

6.33 Pullbacks. A pullback of functions \( X \xrightarrow{f} Z \) and \( Y \xrightarrow{g} Z \) consists of a set \( X \times_Z Y \) and functions \( X \times_Z Y \xrightarrow{\pi_0} X \) and \( X \times_Z Y \xrightarrow{\pi_1} Y \) for which \( f \circ \pi_0 = g \circ \pi_1 \) and enjoys the property that if \( f \circ u = g \circ v \), then there is a unique \( \varphi \) for which \( \pi_0 \circ \varphi = u \) and \( \pi_1 \circ \varphi = v \).

![Diagram of pullback](image)

A pullback of functions is given by:

\[
X \times_Z Y = \{(x, y) \in X \times Y \mid f(x) = g(y)\}
\]

In the quantum computing literature, each point \( (x, y) \) of the pullback is called a claw of the functions \( f \) and \( g \) [33], hence, a pullback is the set of all claws of two functions.

Since a pullback is a limit 2.553, it is also a right Kan extension by 2.816.

6.331 To calculate the pullback we may first order the set of values \( \{f(x) \in Y \mid x \in X\} \) and then for each \( y \in Y \), search for an \( x \) with \( f(x) = g(y) \). The sorting takes \( O(X \log X) \) comparisons and, for each \( y \), the search requires \( O(\log X) \) comparisons. The classical complexity of calculating a pullback is, therefore, \( O(n \log X) \) where \( n = \max\{X, Y\} \). See [33].

6.34 Coproducts. In a category \( \mathcal{C} \), a coproduct of objects \( X \) and \( Y \) is a triple \((X + Y, i_X, i_Y)\) with \( i_X : X \to X + Y \) and \( i_Y : Y \to X + Y \) and which enjoys the property that if \((Z, u_X, u_Y)\) is any such triple, then there is a unique morphism \( \varphi : X + Y \to Z \) for which the following commutes.

![Diagram of coproduct](image)

This is the universal mapping property defined in 2.63.
A coproduct is a particular kind of colimit and, by 2.826, colimits may be computed as left Kan extensions. Moreover, all left Kan extensions may be computed from two particular kinds of colimits: coproducts and coequalizers. As discussed at the start of this chapter, this follows from 2.824 and 2.661. If we unwrap the layers in these theorems, we find that coproducts are expressed as the following left Kan extension.

\[
\begin{array}{ccc}
\{0,1\} & \xrightarrow{!} & \{\ast\} \\
\downarrow & & \downarrow \\
(X,Y) & \xrightarrow{\langle i_0,i_1\rangle} & X\coprod Y \\
\end{array}
\]

We know from 3.132 that \(\text{Set}\) has all small coproducts, hence, it has binary coproducts. Disjoint union gives one solution to this universal mapping property.

By 3.2 we know that the category of finite sets is equivalent to the category \(\Delta\) of simplicial sets, hence, in working with a finite set \(X\) with \(|X| = n\), we may instead work with its representative \(\{0, \ldots, n-1\}\) in \(\Delta\). From this we obtain a simple formula for categorical coproducts of finite sets: If \(X = \{0, \ldots, m-1\}\) and \(Y = \{0, \ldots, n-1\}\), then

\[
X + Y = \{0, \ldots, m + n - 1\} \quad i_X(x) = x \quad \text{and} \quad i_Y(y) = y + m.
\]

That is, a coproduct of ordinals is a sum (with two inclusion maps). If \(X\) and \(Y\) are finite and have cardinalities \(|X|\) and \(|Y|\), then \(X + Y\) has cardinality \(|X + Y|\).

6.341 [50] presents classical circuits for adding binary integers and analyzes the time and space resources that they require. The ripple-carry adder can add two \(n\)-bit integers in \(\Theta(n)\) time using a circuit with size \(\Theta(n)\). The carry-lookahead adder can perform the operation in \(\Theta(\lg(n))\) time using a circuit that is also of size \(\Theta(n)\). Given the observations in 6.34, we see that these give classical circuits for computing an important class of left Kan extension. It follows from 2.824 and 2.661 that left Kan extensions can be computed from two particular kinds of left Kan extension one of which is linear in the size of its input. Of course the size of this input is not known a priori from the inputs to a general left Kan extension calculation.

6.35 Coequalizers. A coequalizer of functions \(X \xrightarrow{f} Y\) and \(X \xrightarrow{g} Y\) is a pair \((Q, q)\) for which \(Y \xrightarrow{q} Q\) and \(q \circ f = q \circ g\) and which enjoys the property that, if \((Q', q')\) is any such pair, then there is a unique \(Q \xrightarrow{\varphi} Q'\) for which \(\varphi \circ q = q'\).

\[
\begin{array}{ccc}
X & \xrightarrow{f} & Y \\
\downarrow & & \downarrow q \\
Q & \xrightarrow{q'} & Q' \\
\end{array}
\]

This is the universal mapping property described in 2.64. From the proof of 3.131, we have the following formula for a coequalizer of functions.

\[
Q = Y/\sim \quad \text{and} \quad q(y) = [y].
\]

where \(\sim\) is the equivalence relation generated by all \(f(x) \sim g(x)\).
To understand algorithms for coequalizers (of functions between sets), it is useful to construe \( f : X \rightarrow Y \) and \( g : X \rightarrow Y \) as inducing a graph that has \( Y \) as its set of nodes and an edge \((f(x), g(x))\) for each \( x \in X \). To compute a coequalizer, we must compute the connected components of this graph and then assign each vertex to its component. Alternatively, compute the transitive closure of the binary relation expressed by the graph then assign vertices to equivalence classes. There are several algorithmic formulations of this calculation. [50] and [117] describe the union-find family of algorithms including Quick-Union, Quick-Find and related concepts such as path compression and disjoint-set forests. Alternatively, [24] describes the dominating set problem which we discuss in 7.221 on page 101. A coequalizer of two functions \( f : X \rightarrow Y \) and \( g : X \rightarrow Y \) computes the dominating set of the transitive closure of the graph that has \( Y \) as its vertex sets and has edges \((f(x), g(x))\) for all \( x \in X \).

6.36 Morphisms of a Category. If \( B \) is a finitely-presented category, \( A \) is the discrete category having no generators and \(|A| = |B|\), \( F : A \rightarrow B \) is the inclusion functor, and \( X : A \rightarrow \text{Set} \) is \( X(A) = \) a singleton set, then the left Kan extension of \( X \) along \( F \) gives the set of morphisms of \( B \).

For example, consider the figure below in which \( B \) has two objects \( P \) and \( Q \), two generators \( p \) and \( q \) and one relation \( p \circ q \circ p \circ q = \text{id} \). The resulting category has nine morphisms including the two generating morphisms. The category is shown below on the right. The tables that occur in the Carmody-Walters algorithm are also shown below.
6.37 Orbits of Discrete Iterators. A discrete iterator is a set $S$ equipped with a function $f : S \to S$. We construe $f$ as defining a dynamics on $S$. The term discrete refers to the fact that $S$ is a set rather than a set equipped with a topology or $\sigma$-algebra. The term iterator indicates that time evolves in discrete steps.

Figure 6.2: Example of a discrete iterator. The set $S$ has 11 elements. The arrows in the diagram indicate the action of the function $f : S \to S$.

Discrete iterators are actions defined on a finitely-presented category $\mathbb{N}$ that has one object, one generator and no relations. See Figure 6.3.

Figure 6.3: Generators and relations for categories used to compute orbits of discrete iterators by left Kan extension. The category $\mathbb{N}$ has one object, one generator, and no relations. $\mathcal{B}$ has one object but neither generators nor relations. Its only morphism is an identity. The functor $F$ maps $\bullet$ to $\ast$ and maps the generator of $\mathbb{N}$ to the empty path (i.e., the identity morphism) in $\mathcal{B}$.

For the calculation of orbits of discrete iterators, the Carmody-Walters algorithm takes a simplified form. Since the category $\mathcal{B}$ has no generators, there are no $L$-tables or relation tables to keep track of. The fact that $\mathbb{N}$ has only one object implies that the $\epsilon$-table has only a single column of unknown values and the naturality table has only two.

Figure 6.4: Discrete iterator from Figure 6.2 with states numbered for use in the Kan extension algorithm.
6.38 The Todd-Coxeter Coset Enumeration Procedure. The cosets of a subgroup \( H \) of a given group \( G \) can be calculated as a Kan extension. Moreover, the Todd-Coxeter coset enumeration procedure discussed in 4.21 is a special case of the Carmody-Walters algorithm for computing Kan extensions.

Recall that any group \( G \) may be construed as a category \( G \) having a single element; which we denote \( * \). The morphisms of \( G \) are the elements of \( G \). In particular, the identity element of the group corresponds to the identity morphism for the object \( * \). Composition \( \circ \) in \( G \) is given by the group operation: \( g' \circ g = g' \cdot g \). Associativity and the identity axioms for \( \circ \) follow from the corresponding properties of the group. Any group homomorphism \( \varphi : G \to G' \) induces a functor \( G \to G' \) via \( \ast \mapsto \ast \) on objects and \( g \mapsto \varphi(g) \) on morphisms.

A presentation of a group \( G \) by generators and relations induces a presentation of the corresponding category \( G \). If, for example, \( G \) has presentation \( (s, t \mid s^3 = t^2 = 1, t s t = s^2) \), then \( G \) has the following presentation: \( G = (s, s^{-1}, t, t^{-1} \mid s^3 = t^2 = 1, t s t = s^2, s s^{-1} = s^{-1} s = t t^{-1} = t^{-1} t = 1) \). In general, we obtain a category presentation from a group presentation by adding a generator \( g^{-1} \) and relations \( g g^{-1} = g^{-1} g = 1 \) for each generator of the group.

Let \( H \) be a subgroup of a group \( G \) and let \( i : H \to G \) be the inclusion group homomorphism (i.e., \( i(h) = h \) for all \( h \in H \)). As described above, \( i \) induces a functor \( H \to G \) between the associated one-object categories. Define a functor \( X : H \to \text{Set} \) by \( H(\ast) = 1 \) where 1 is any one-point set (e.g., \( 1 = \{0\} \)). The left Kan extension \( L : G \to \text{Set} \) of \( X \) along \( F \) gives the representation of \( G \) on the (right) cosets of \( H \).

\[
\begin{array}{ccc}
H & \xrightarrow{i} & G \\
X & \searrow & L \\
& \swarrow & \downarrow \\
& \text{Set} & \\
\end{array}
\]

That is, \( L(\ast) = \{1, \ldots, n\} \) where \( n = [G : H] \) (the number of cosets of \( H \) in \( G \)). The coset \( H \) contains the identity element of \( G \) and is recovered as \( \epsilon_\ast(0) \) (where \( \epsilon_\ast : X(\ast) \to L(\ast) \) is the one component of the natural transformation \( \epsilon \)). and \( L(g) \) gives a permutation of the cosets.
Chapter 7

Quantum Algorithms for Computing Kan Extensions

In order to ensure that we made progress on the challenging task of developing quantum algorithms for computing Kan extensions, we have relied on theorems to break the problems into a simpler steps. 2.813 demonstrates that any right Kan extension may be computed as limit of a functor constructed from the ingredients used to specify it. From this and 2.564 it follows that all right Kan extensions can be computed from two particular kinds of limits: products and equalizers. In seeking a quantum algorithm for computing right Kan extensions, we first seek quantum algorithms for products and equalizers. Similarly, 2.824 demonstrates that any left Kan extension may be computed as colimit while 2.661 computes arbitrary colimits from two particular kinds: coproducts and coequalizers. In seeking a quantum algorithm for computing left Kan extensions, we first seek quantum algorithms for coproducts and coequalizers.

7.1 Right Kan Extensions

In 6.3, we discussed several examples of left and right Kan extensions and classical algorithms for computing them. In the following subsections we discuss quantum algorithms for these special cases.

7.11 Products. As discussed in 6.31, a product of sets $X$ and $Y$ is defined by a universal mapping property 2.53, is a particular kind of limit, hence, by 2.816, may be computed as a right Kan extension. Moreover 2.813 and 2.564 establish that all right Kan extensions may be computed from two particular kinds of limits: products and equalizers. In order to develop quantum algorithms for computing right Kan extensions in general, we first seek quantum algorithms for products and equalizers.

We know from 3.12 that $\text{Set}$ has all products. By 3.2 we know that the category of finite sets is equivalent to the category $\Delta$ of simplicial sets, hence, in working with a finite set $X$ with $|X| = n$, we may instead work with its representative $\{0, \ldots, n-1\}$ in $\Delta$. From this we obtain a simple formula for categorical products of finite sets: If $X = \{0, \ldots, m-1\}$ and $Y = \{0, \ldots, n-1\}$, then

$$X \times Y = \{0, \ldots, (m \times n) - 1\} \quad \pi_0(k) = \lfloor m/n \rfloor \quad \text{and} \quad \pi_1(k) = k \mod n$$

That is, a product of ordinals is given by multiplication.
Quantum algorithms for computing products are known [126]. These give quantum algorithms for computing particular right Kan extensions.

**Equalizers.** Given morphisms \( f, g : X \to Y \) in a category \( \mathcal{C} \), an equalizer of \( f \) and \( g \) is a pair \((E, e)\) with \( E \) an object and \( e : E \to X \) a morphism for which \( f \circ e = g \circ e \). Given any such pair \((E', e')\), there is a unique \( \varphi : E' \to E \) for which \( e' = e \circ \varphi \) as shown below.

\[
\begin{array}{c}
e' \\
\downarrow \\
E' \\
\downarrow \\
X \\
\downarrow \\
Y \\
\end{array} \quad \begin{array}{c}
e \\
\downarrow \\
E \\
\downarrow \\
X \\
\downarrow \\
Y \\
\end{array} \quad \begin{array}{c}
f \\
\downarrow \\
Z \\
\end{array} \quad \begin{array}{c}
g \\
\downarrow \\
Z \\
\end{array}
\]

This is the universal mapping property defined in 2.54. If \( \mathcal{C} = \text{Set} \), then 
\[
E = \{ x \in X \mid f(x) = g(x) \}
\]
and \( e : E \to X \) is the inclusion function as shown in 3.121. The subset \( E \) of \( X \) is also determined by the characteristic function \( \chi : X \to \{0, 1\} \) where
\[
\chi(x) = \begin{cases} 
1 & \text{if } f(x) = g(x) \\
0 & \text{otherwise.}
\end{cases}
\]
The size of \( E \) is not known a priori, however.

**Pullbacks.** A pullback of functions \( X \overset{f}{\to} Z \) and \( Y \overset{g}{\to} Z \) consists of a set \( X \times_Y Y \) and functions \( X \times_Y Y \overset{\pi_0}{\to} X \) and \( X \times_Y Y \overset{\pi_1}{\to} Y \) for which \( f \circ \pi_0 = g \circ \pi_1 \) and enjoys the property that if \( f \circ u = g \circ v \), then there is a unique \( \varphi \) for which \( \pi_0 \circ \varphi = u \) and \( \pi_1 \circ \varphi = v \).

\[
\begin{array}{c}
u \\
\downarrow \\
\pi_1 \\
\downarrow \\
Y \\
\downarrow \\
Z \\
\end{array} \quad \begin{array}{c}
u \\
\downarrow \\
\pi_0 \\
\downarrow \\
X \\
\downarrow \\
X \times Z \\
\end{array} \quad \begin{array}{c}
T \\
\end{array}
\]

A pullback of functions is given by:
\[
X \times_Y Y = \{ (x, y) \in X \times Y \mid f(x) = g(y) \}
\]
7.131 In the quantum computing literature, as discussed in 6.33, each point \((x, y)\) of the pullback is called a **claw** of the functions \(f\) and \(g\) [33], hence, a pullback is the set of all claws of two functions.

7.132 *Quantum computers can not improve upon classical (probabilistic) complexity of exact pullback calculations.*

*Because:* such an algorithm would contradict Theorem 6.1 of [33]. Assume we have a quantum algorithm that calculates pullbacks. Given \(f : X \rightarrow Y\), form the pullback \(P\) of \(f\) with itself. \(P - X\) counts the number of **collisions** (i.e., remove the diagonal). Consequently, an efficient pullback algorithm gives an an efficient algorithm for exactly counting the number of collisions. This gives the desired contradiction.

7.133 *Quantum computers can not improve upon classical (probabilistic) complexity of exact equalizer calculations.*

*Because:* algorithms for equalizers and products give an algorithm for pullbacks. Specifically, given functions \(f : X \rightarrow Z\) and \(g : Y \rightarrow Z\) we may first compute the product \((X \times Y, \pi_1, \pi_2)\) then compute the equalizer of the composites \(f \circ \pi_1\) and \(g \circ \pi_2\).

![Diagram](https://via.placeholder.com/150)

The complexity of the algorithm is determined by that of the equalizer construction.

7.134 As the analyses above suggest, pullbacks are related to Grover’s Search Algorithm. Given the following diagram of sets and functions

![Diagram](https://via.placeholder.com/150)

with 1 a one-point set (i.e., a terminal object in Set), then the function \(y\) selects an element, that we will also call \(y\), of \(Y\). A pullback of this diagram is given by

![Diagram](https://via.placeholder.com/150)

where \(f^{-1}(y)\) is the inverse image of \(y \in Y\) under the action of \(f\). That is, \(f^{-1}(y) = \{x \in X \mid f(x) = y\}\). In particular, if \(Y = 2 = \{0, 1\}\) is a two-point set, and \(y = 1\), then \(f\) is the characteristic function of the set \(f^{-1}(1)\). Grover’s Search Algorithm finds an element \(x^*\) of this set. If we construe this element as a
function $1 \rightarrow f^{-1}(y)$, then, we see that Grover’s Algorithm

That is, an algorithm for computing pullbacks gives an algorithm for solving Grover search.

### 7.14 General Right Kan Extensions

The discussion in the previous subsections establishes the following.

*Quantum computers can not improve upon classical (probabilistic) complexity of exact right Kan extension calculations.*

*Because:* this follows from 7.132 and 7.133.

### 7.2 Left Kan Extensions

In 6.3, we discussed several examples of left and right Kan extensions and classical algorithms for computing them. In the following subsections we discuss quantum algorithms for these special cases.

#### 7.21 Coproducts

In a category $\mathcal{C}$, a coproduct of objects $X$ and $Y$ is a triple $(X + Y, i_X, i_Y)$ with $i_X : X \rightarrow X + Y$ and $i_Y : Y \rightarrow X + Y$ and which enjoys the property that if $(Z, u_X, u_Y)$ is any such triple, then there is a unique morphism $\varphi : X + Y \rightarrow Z$ for which the following commutes:

which is the universal mapping property defined in 2.63.

A coproduct is a particular kind of colimit and, by 2.826, colimits may be computed as left Kan extensions. Moreover, all left Kan extensions may be computed from two particular kinds of colimits: coproducts and coequalizers. As discussed at the start of this chapter, this follows from 2.824 and 2.661. If we unwrap the layers in these theorems, we find that coproducts are expressed as the following left Kan extension.

We know from 3.132 that $\text{Set}$ has all small coproducts, hence, it has binary coproducts. Disjoint union gives one solution to this universal mapping property.
By 3.2 we know that the category of finite sets is equivalent to the category $\Delta$ of simplicial sets, hence, in working with a finite set $X$ with $|X| = n$, we may instead work with its representative $\{0, \ldots, n - 1\}$ in $\Delta$. From this we obtain a simple formula for categorical coproducts of finite sets: If $X = \{0, \ldots, m - 1\}$ and $Y = \{0, \ldots, n - 1\}$, then

$$X + Y = \{0, \ldots, m + n - 1\} \quad i_X(x) = x \quad \text{and} \quad i_Y(y) = y + m$$

That is, a coproduct of ordinals is a sum (with two inclusion maps). If $X$ and $Y$ are finite and have cardinalities $|X|$ and $|Y|$, then $X + Y$ has cardinality $|X + Y|$. Quantum algorithms for addition are known: $[16, 51, 58, 59, 126]$. These give quantum algorithms for computing particular left Kan extensions.

### 7.22 Coequalizers

A coequalizer of functions $X \xrightarrow{f} Y$ and $X \xrightarrow{g} Y$ is a pair $(Q, q)$ for which $Y \xrightarrow{q} Q$ and $q \circ f = q \circ g$ and which enjoys the property that, if $(Q', q')$ is any such pair, then there is a unique $Q \xrightarrow{\varphi} Q'$ for which $\varphi \circ q = q'$.

This is the universal mapping property described in 2.64. From the proof of 3.131, we have the following formula for a coequalizer of functions.

$$Q = Y/\sim \quad \text{and} \quad q(y) = [y].$$

where $\sim$ is the equivalence relation generated by all $f(x) \sim g(x)$.

### 7.221 Let $G = (V, E)$ be a graph. A dominating set for $G$ is a set $V' \subset V$ of vertices with the property that if $v \in V/V'$, then there is a vertex $v' \in V'$ and an edge $\{v, v'\} \in E$. One way to view a coequalizer of functions $f : X \to Y$ and $g : X \to Y$ is as a dominating set for the graph that has $Y$ as its vertex set and the transitive closure of $\{(f(x), g(x)) \mid x \in X\}$ as its edge set. Elements of the dominating set for this graph are representatives of distinct equivalence classes of the coequalizer $Q$. The general dominating set problem is NP-complete. Coequalizers are a particular subclass of problems.

Lemma 11 of [24] gives conditions under which the dominating set problem is not solvable. Since a coequalizer of two functions always exists by 3.131, coequalizer problems do not satisfy the condition of that lemma. Lemma 10 of [24] characterizes another class of dominating set problems which are solvable. By Theorem 16 of [24], the quantum query complexity of the dominating set problem is $\Omega(n^{1.5})$. By Theorem 17 of [24], this lower bound cannot be improved by Ambainis’ method (see [11]).
Chapter 8

Java Implementation

In order to reinforce our understanding classical algorithms and prepare to develop quantum algorithms, we first implemented classical Kan extension algorithms in the Java programming language. We chose Java since it is the language with which we have the most development experience. The extensive libraries available in Java and its platform independence were also desirable features. For complex number and matrix calculations, for example, we used the Apache Commons mathematics library\(^1\). The Java documentation is available in the directory java/jdoc/index.html that is included in the final contract deliverables.

8.1 Software Design

To simplify the development of data input and output code, we developed XML schema for the data types to be read and written. We then used the JAXB compiler to automatically generate the Java code to parse the data files. Figure 8.1 shows the design of our Java implementation.

\[^1\text{See}\,\text{http://commons.apache.org/math.}\]
8.2 Input and Output Files

We use XML files for input to and output from the algorithms we implement. These files are used for exchange of category, functor, action and Kan extension data. For each of these structures, we define an XML schema and use the JAXB compiler xjc to generate Java classes that marshal and unmarshal corresponding instance documents. We then create a class with static methods for converting the JAXB objects to and from objects of classes of our other packages.

8.2.1 Categories. We define the XML schema category-schema.xsd for exchange of finitely-presented categories between our applications and data files. The schema is shown in Listing 3.

Each instance document has a top-level <category> element followed by <name>, <objects>, <generatorList> and <relationList> elements. The <name> element is a string that describes the category. The <objects> element is an integer that indicates the number of objects in the category. If this value is \( n \) then the names 0, 1, \ldots, \( n - 1 \) are used for these objects. Subsequent implementations of category-schema.xsd could support a list of string names.

The <generatorList> element contains zero or more <generator> elements. These are the generators of the category. Each <generator> element has <domain> and <codomain> elements. These refer to integer indices of the domain and codomain objects. Note that even if there are no <generator> elements, the instance document must contain an (empty) <generatorList> element.

The <relationList> element contains zero or more <relation> elements. Each <relation> element consists of two <path> elements corresponding to the two sides of the equation in a relation for the finitely-presented category. Each <path> element is a list of integer indices of the generators. For example, if \( c \circ b \circ a = z \circ y \circ x \) is a relation, then the <path> elements would be \( a \ b \ c \) and \( x \ y \ z \). Note the reverse ordering from that in the composition notation. Even if there are no <relation> elements, the instance document must contain an (empty) <relationList> element.

An instance document for the schema shown in Listing 3 is shown in Listing 4.
Listing 3: XML schema category-schema.xsd for describing finitely-presented categories
```xml
<?xml version="1.0" encoding="UTF-8" ?>
<category xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="category-schema.xsd">
  <name>Simple example</name>
  <objects>2</objects>
  <generatorList>
    <generator>
      <name>f</name>
      <domain>0</domain>
      <codomain>1</codomain>
    </generator>
    <generator>
      <name>g</name>
      <domain>1</domain>
      <codomain>0</codomain>
    </generator>
  </generatorList>
  <relationList>
    <relation>
      <path>
        <leg>f</leg>
        <leg>g</leg>
      </path>
    </relation>
    <relation>
      <path>
        <leg>g</leg>
        <leg>f</leg>
      </path>
    </relation>
  </relationList>
</category>
```


Listing 4 includes an instance document for the XML schema `category-schema.xsd`. The finitely-presented category that it describes is shown in Figure 8.2. This category has two objects, 0 and 1, and two generators, which are also labeled as 0 and 1. The category has two relations which indicate that the composition of either generator with the other is an identity morphism. Note that the composite

\[
\begin{array}{ccc}
0 & \xrightarrow{0} & 1 & \xrightarrow{1} & 0 \\
\end{array}
\]

is written \(1 \circ 0\) in composition notation but is expressed by the `<path>` element 0 1 on line 21 of Listing 4. Identity morphisms are expressed by empty `<path>` elements as on lines 22 and 26 of Listing 4.

Figure 8.2: Generators and relations for the finitely-presented category described in the XML document shown in Listing 4. The first `<path>` in the XML document is \(1 \circ 0\) (note the order reversal) and the second is the empty path, hence, the first `<relation>` element is \(1 \circ 0 = id_0\).
8.22 Functors. We define the XML schema `functor-schema.xsd` for exchange of data involving functors between finitely-presented categories. The schema is shown in Listing 5.

Each instance document has a top-level `<functor>` element followed by `<name>`, `<objectImages>` and `<generatorImages>` elements. The `<name>` element is a string that describes the functor. The `<objectImages>` element is a sequence of `<objectImage>` elements.
Listing 5: XML schema `functor-schema.xsd` for describing functors between finitely-presented categories
8.23 Actions. We define the XML schema action-schema.xsd for exchanging data about $\text{Set}$-valued functors defined on finitely-presented categories. The schema is shown in Listing 6. Let $X : \mathcal{A} \to \text{Set}$ be an action.

Each instance document has a top-level <action> element followed by <name>, <objectImages> and <generatorImages> elements. The <name> element is a string that describes the action. The <objectImages> element is a sequence of <objectImage> elements. Each of <objectImage> element contains a <input> and an <output>. The former contains an integer index $A$ of a domain object. The latter contains an integer that specifies the size $n$ of the image $X(A)$ under the action. We assume that $X(A) = \{0, 1, \ldots, n - 1\}$.

The <generatorImages> element contains zero or more <generatorImage> elements each of which consists of a <generator> element, which is an integer $a$ index of a generator in the domain category, and a <map> element, which is a sequence of integers that define the function $X(a)$. Entry $i$ in the sequence is the integer $X(a)(i)$.

Listing 7 shows an instance document of action-schema.xsd. It describes an action on the monoid of natural numbers. Such actions are discrete iterators and are discussed further in Section 6.37 on page 94.
Listing 6: XML schema `action-schema.xsd` for describing actions defined on finitely-presented categories.
Listing 7 shows an action-schema.xsd document that describes a discrete iterator, that is, a Set-valued functor defined on the monoid of natural numbers.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<action xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:noNamespaceSchemaLocation="action-schema.xsd">
  <name>Discrete iterator with 11 states, four orbits, two fixed points, a two-cycle
       and a three-cycle</name>
  <objectImages>
    <objectImage>
      <input>0</input>
      <output>11</output>
    </objectImage>
  </objectImages>
  <generatorImages>
    <generatorImage>
      <generator>0</generator>
      <map>1 2 2 4 3 6 6 8 9 10 8</map>
    </generatorImage>
  </generatorImages>
</action>
```

Listing 7: Instance document for the XML schema action-schema.xsd defined on the monoid of natural numbers. X is a discrete iterator shown in Figure 6.4.

### 8.3 Java Packages

Our Java implementations are organized into packages. These packages and the classes defined in them can be explored using the Java documentation that is available in the directory java/jdoc/index.html that is included in the final contract deliverables. The packages include the following:

- **org bmrc qkan classical**
  Provides the classes for computing Kan extensions by the classical algorithm.

- **org bmrc qkan common**
  Provides the common classes for computing Kan extensions via the classical and quantum algorithms.

- **org bmrc qkan common xml**
  Provides the classes for using JAXB-generated code to marshal and unmarshal XML documents.

- **org bmrc qkan common xml action**
  JAXB-generated code for managing XML documents that utilize action-schema.xsd.

- **org bmrc qkan common xml category**
  JAXB-generated code for managing XML documents that utilize category-schema.xsd.

- **org bmrc qkan common xml functor**
  JAXB-generated code for managing XML documents that utilize functor-schema.xsd.

- **org bmrc qkan quantum**
  Provides the classes for working with qubits and systems of qubits.

- **org bmrc qkan test classical**
  Provides classes for testing classes in the org bmrc qkan classical package.

- **org bmrc qkan test common**
Provides classes for testing classes in the org.bmrc.qkan.common package.

- **org.bmrc.qkan.test.quantum**
  Provides classes for testing classes in the org.bmrc.qkan.quantum package.

### 8.4 Examples

To test the classes of our implementation, we developed a hierarchy of testing packages that mirror our hierarchy of application packages. Classes in the latter have corresponding classes in the former that are used to test functionality.

#### 8.4.1 Orbits

Listing 8 shows a category-schema.xsd document that describes the monoid of natural numbers.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<category xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="category-schema.xsd">
  <name>Monoid of natural numbers</name>
  <objects>1</objects>
  <generatorList>
    <generator>
      <name>f</name>
      <domain>0</domain>
      <codomain>0</codomain>
    </generator>
  </generatorList>
</category>
```

Listing 8: XML document nat.xml implementing the finitely-presented category \( \mathbb{N} \) shown in Figure 6.3.

Listing 9 shows a category-schema.xsd document that describes a category with a single morphism (hence, a single object).

```xml
<?xml version="1.0" encoding="UTF-8"?>
<category xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="category-schema.xsd">
  <name>Monoid of natural numbers</name>
  <objects>1</objects>
  <generatorList>
    <generator>
      <name>f</name>
      <domain>0</domain>
      <codomain>0</codomain>
    </generator>
  </generatorList>
</category>
```

Listing 9: XML document one.xml implementing the finitely-presented category \( \mathbb{B} \) shown in Figure 6.3.
Listing 10 shows a functor-schema.xsd document that describes a functor from the natural numbers to the one-object category.

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<functor xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="functor-schema.xsd">
    <name>Functor from the monoid of natural numbers to a one-object category</name>
    <objectImages>
        <objectImage>
            <input>0</input>
            <output>0</output>
        </objectImage>
    </objectImages>
    <generatorImages>
        <generator>f</generator>
        <path/></path>
    </generatorImages>
</functor>
```

Listing 10: XML document F.xml implementing the functor $F$ shown in Figure 6.3.

### 8.42 Coset Enumeration

Appendix B shows sample input XML files and output used to apply the Carmody-Walters algorithm to enumerate cosets of a subgroup of a finitely-presented group.
Chapter 9

Quantum Programming Languages

In this Chapter we discuss use of quantum programming languages to implement algorithms for computing Kan extensions. We focus on two languages, Quipper and QuaFL, that are underdevelopment through the IARPA Quantum Computer Science program. Since both languages employ a functional programming paradigm and, in particular, Quipper is embedded in the Haskell programming language, we first discuss use of Haskell to implement algorithms for computing Kan extensions.

9.1 Haskell Implementation

2.813 demonstrates that any right Kan extension may be computed as limit of a functor constructed from the ingredients used to specify it. From this and 2.564 it follows that all right Kan extensions can be computed from two particular kinds of limits: products and equalizers. Similarly, 2.824 demonstrates that any left Kan extension may be computed as colimit while 2.661 computes arbitrary colimits from two particular kinds: coproducts and coequalizers. Haskell implementations of particular Kan extensions are interspersed throughout this chapter and are integrated into modules that are included in Appendix C.

9.11 Right Kan Extensions. In seeking a quantum algorithm for computing right Kan extensions, we first seek quantum algorithms for products and equalizers.

9.11.1 Terminal Objects. The universal mapping property that defines terminal objects in a category is stated in 2.52. In the category $\text{Set}_f$ of finite sets, or the category $\Delta$ of ordinals, any one element set 1 is a terminal object. Given any set $X$, there is a unique function $!_X : X \rightarrow 1$.

Listing 11 gives a Haskell implementation of terminal objects in the simplicial category $\Delta$. We first define the Function type to be ([Int], Int). In $\Delta$, each object is a finite set of the form $X = \{0, 1, \ldots, n-1\}$. To define a function $f : X \rightarrow Y$, it is sufficient to give a list of values $(f(0), f(1), \ldots, f(n-1))$ and to specify the codomain $Y$. If $Y$ were not specified, then the list $[0, 1, 1, 0]$ could represent functions $4 \rightarrow 2$, $4 \rightarrow 3$ or, in fact, $4 \rightarrow n$ for any $n \geq 2$. The Haskell type implements this fact: a function in $\Delta$ is a list of output values together with a specified codomain.

The type UFUnctio in Listing 11 implements the universal function $!$. Given a set $X$, which we implement as an Int in Haskell, $!_X$ is a function $X \rightarrow 1$. We implement this as a UFUnctio named shriek.
CHAPTER 9. QUANTUM PROGRAMMING LANGUAGES

type Function = ([Int], Int)

type UFunction = Int -> Function

--- Any singleton set 1 is a terminal object in the category of finite sets
--- (or the category of sets). This object satisfies a universal mapping
--- property: given any set X, there is a unique function X -> 1.

terminalObject :: (Int, UFunction)
terminalObject = (1, shriek)
where shriek = (n -> ((take n (repeat 1)), 1))

Listing 11: Terminal object implemented in Haskell

Here are examples of how to use this implementation in ghci, the interactive Haskell interpreter.

> fst terminalObject
1

and

> (snd terminalObject) 3
([1,1,1],1)

where the latter output represents the function 3 -> 1.

9.112 Products. The universal mapping property defining products in a category is stated in 2.53.
By 3.2 we know that the category of finite sets is equivalent to the category ∆ of simplicial sets, hence,
in working with a finite set X with |X| = n, we may instead work with its representative {0, ..., n - 1}
in ∆. In 7.11 we obtained a simple formula for categorical products of finite sets: If X = {0, ..., m - 1} and Y = {0, ..., n - 1}, then

\[ X \times Y = \{0, \ldots, (m \times n) - 1\} \quad \pi_0(k) = \lfloor m/n \rfloor \quad \text{and} \quad \pi_1(k) = k \mod n \]

That is, a product of ordinals is given by multiplication.

In Listing 12 we give a Haskell implementation of products in the simplicial category ∆. The type Function of functions was defined in Listing 11. Note that we name the construction product' rather
than product so as not to conflict with a standard library function in Haskell that has the latter name.
For simplicity, we do not show the implementation of the universal mapping property that characterizes products.

--- A product of two sets X and Y consists of a set X x Y and two functions
--- \( \pi_0 : X \times Y \to X \) and \( \pi_1 : X \times Y \to Y \) that satisfy a universal mapping
--- property. One solution is given by cartesian product. In the case of
--- skeletal sets (i.e., sets of the form \{0, 1, ..., n-1\}), multiplication
--- gives the product operation.

product' :: Int -> Int -> (Int, (Function, Function))
product' x y = (xy, (pi0, pi1))
where xy = x*y
    pi0 = ([div m y | m <- [0..(xy-1)]], x)
    pi1 = ([mod k y | k <- [0..(xy-1)]], y)

Listing 12: Products implemented in Haskell
Here is an example executed in ghci.

\[ \texttt{> product' 2 3} \]
\[ (6,([0,0,0,1,1,1],2),([0,1,2,0,1,2],3)) \]

The output gives the product object, namely 6, together with the two projections 6 \rightarrow 2 and 6 \rightarrow 3.

### 9.1.13 Equalizers

The universal mapping property defined by equalizers 7.12 is stated in 2.54. In 7.12 we obtained a simple formula for coequalizers of functions:

\[ E = \{ x \in X \mid f(x) = g(x) \} \]

and \( e : E \rightarrow X \) is the inclusion function as shown in 3.121.

In Listing 13 we give a Haskell implementation of equalizers in the simplicial category \( \Delta \). The type `Function` of functions was defined in Listing 11. Note that the Haskell list comprehension syntax allows us to give an implementation that is almost identical to the mathematical definition.

```haskell
-- An equalizer of two functions \( f : X \rightarrow Y \) and \( g : X \rightarrow Y \) is a function
-- that satisfies a universal mapping property. One solution
-- is given by \( X' = \{ x \in X \mid f(x) = g(x) \} \).

equalizer :: Function -> Function -> Function

where x = length (fst f) -- the domain of f and g

  y = snd f

  f' = fst f

  g' = fst g

  e = [i | i < [0..(x-1)], (f' !! i) == (g' !! i)]
```

Listing 13: Equalizers implemented in Haskell

If we define the following in a Haskell script

\[
\begin{align*}
f & : \text{Function} \\
f & = ([0,1,2],4) \\
g & : \text{Function} \\
g & = ([0,2,2],4)
\end{align*}
\]

then we can use ghci to test our equalizer implementation:

\[ \texttt{> equalizer f g} \]
\[ ([0,2],3) \]

That is, if \( f : 3 \rightarrow 4 \) and \( g : 3 \rightarrow 4 \) are the functions defined above, then the equalizer of \( f \) and \( g \) is the function \( e : 2 \rightarrow 3 \) for which \( e(0) = 0 \) and \( e(1) = 2 \). This is the case since \( f(0) = g(0) \) and \( f(2) = g(2) \) but \( f(1) \neq g(1) \).
9.114 **Pullbacks.** The universal mapping property defined by pullbacks in a category is stated in 2.55. A pullback of functions \( X \xrightarrow{f} Z \) and \( Y \xrightarrow{g} Z \) consists of a set \( X \times_Z Y \) and functions \( X \times_Z Y \xrightarrow{\pi_0} X \) and \( X \times_Z Y \xrightarrow{\pi_1} Y \) for which \( f \circ \pi_0 = g \circ \pi_1 \) and enjoys the property that if \( f \circ u = g \circ v \), then there is a unique \( \varphi \) for which \( \pi_0 \circ \varphi = u \) and \( \pi_1 \circ \varphi = v \).

A pullback of functions is given by:

\[
X \times_Z Y = \{ (x, y) \in X \times Y \mid f(x) = g(y) \}
\]

and we can apply this in the simplicial category \( \Delta \).

In 7.133 we described how to construct pullbacks from products and equalizers and we used this construction to establish a theorem about quantum algorithms for computing equalizers. In Listing 14 we give a Haskell implementation of pullbacks in the simplicial category \( \Delta \) that exploits this construction. Note that we must first implement composition of our \texttt{Function} data type.

```haskell
-- Composition of functions f:X->Y and g:Y->Z.
compose :: Function -> Function -> Function
compose g f = (gf, z)
  where x = length (fst f)
         z = snd g
         gf = [ (fst g) !! fx | fx <- fst f ]

-- Implementation of a pullback of f:X->Z and g:Y->Z. We use
-- the construction of pullbacks from a product and an equalizer.
pullback :: Function -> Function -> (Function, Function)
pullback f g = (f', g')
  where x = length (fst f)
        y = length (fst g)
        p = product x y
        pi0 = fst (snd p) -- the projection X x Y -> X
        pi1 = snd (snd p) -- the projection X x Y -> Y
        fpi0 = compose f pi0
        gpi1 = compose g pi1
        e = equalizer fpi0 gpi1
        f' = compose pi0 e
        g' = compose pi1 e
```

Listing 14: Pullbacks implemented in Haskell from products and equalizers

If we define the following in a Haskell script

```haskell
f :: Function
f = ([0,1,2],4)

gh :: Function
gh = ([0,2,2],4)
```
then we can use ghci to test our pullback implementation:

> pullback f g
(([0,2,2],3),([0,1,2],3))

That is, if $f : 3 \to 4$ and $g : 3 \to 4$ are the functions defined above, then the pullback of $f$ and $g$ consists of the subset $P = \{(0,0),(1,2),(2,2)\}$ of the cartesian product $2 \times 3 = 6$ together with two projection maps $P \to X$ and $P \to Y$. We can test this in ghci.

> compose f (fst pb)
([0,2,2],4)

> compose g (snd pb)
([0,2,2],4)

This demonstrates the construction that we exploited in 7.133.

9.12 Left Kan Extensions. In seeking a quantum algorithm for computing left Kan extensions, we first seek quantum algorithms for coproducts and coequalizers.

9.121 Initial Objects. The universal mapping property that defines terminal objects in a category is stated in 2.52. In the category $\textbf{Set}_f$ of finite sets, or the category $\Delta$ of ordinals, any one element set 1 is a terminal object. Given any set $X$, there is a unique function $!_X : X \to 1$.

Listing 15 gives a Haskell implementation of initial objects in the simplicial category $\Delta$. The Function data type was defined in Listing 11 to be $([\text{Int}], \text{Int})$. The type $\text{UFunction}$ in Listing 11 implements the universal function $i$. Given a set $X$, which we implement as an Int in Haskell, $i_X$ is a function $0 \to X$.

We implement this as a $\text{UFunction}$ named $\text{coshriek}$.

Here are examples of how to use this implementation in ghci.

> fst initialObject
0

and

> (snd initialObject) 3
([],3)

where the latter output represents the function $0 \to 3$. 

---

--- The empty set $0$ is an initial object in the category of finite sets
--- (or the category of sets). This object satisfies a universal mapping
--- property: given any set $X$, there is a unique function $0 \to X$.

Listing 15: Initial object implemented in Haskell
9.122 Coproducts. The universal mapping property defining coproducts in a category is stated in 6.34. In 3.13 we saw that disjoint union gives coproducts in the category Set. By 3.2 we know that the category of finite sets is equivalent to the category $\Delta$ of simplicial sets, hence, in working with a finite set $X$ with $|X| = n$, we may instead work with its representative $\{0, \ldots, n-1\}$ in $\Delta$. In 6.34 we obtained a simple formula for categorical coproducts of finite sets: $X = \{0, \ldots, m-1\}$ and $Y = \{0, \ldots, n-1\}$, then

$$X + Y = \{0, \ldots, m + n - 1\} \quad i_X(x) = x \quad \text{and} \quad i_Y(y) = y + m.$$ 

That is, a coproduct of ordinals is a sum (with two inclusion maps). If $X$ and $Y$ are finite and have cardinalities $|X|$ and $|Y|$, then $X + Y$ has cardinality $|X + Y|$. That is, a coproduct of ordinals is given by addition.

In Listing 16 we give a Haskell implementation of coproducts in the simplicial category $\Delta$. The type Function of functions was defined in Listing 11. Note that we name the construction coproduct' rather than coproduct so as not to conflict with a standard library function product in Haskell. For simplicity, we do not show the implementation of the universal mapping property that characterizes coproducts.

```
1 1 — A coproduct of two sets X and Y consists of a set X x Y and two functions
2 2 — i0 : X -> X + Y and i1 : Y -> X + Y that satisfy a universal mapping
3 3 — property. One solution is given by disjoint union. In the case of
4 4 — skeletal sets (i.e., sets of the form [0, 1, ..., n-1]), addition
5 5 — gives the product operation.
6 coproduct' :: Int -> Int -> (Int, (Function, Function))
7 coproduct' x y = (x_plus_y, (i0, i1))
8 where x_plus_y = x + y
9    i0 = ([m | m <- [0..(x-1)]], x_plus_y)
10   i1 = ([n+x | n <- [0..(y-1)]], x_plus_y)
```

Listing 16: Coproducts implemented in Haskell

Here is an example executed in ghci.

```
> coproduct' 2 3
(5,([0,1,5],[2,3,4,5]))
```

The output gives the coproduct object, namely 5, together with the two inclusions $2 \to 5$ and $3 \to 5$.

9.123 Coequalizers. The universal mapping property defined by coequalizers is stated in 7.22. In 7.22 we obtained a simple formula for coequalizers of functions: if $f : X \to Y$ and $g : X \to Y$ are functions, then a coequalizer is given by $(q, Q)$ where $Q = Y / \sim$, $\sim$ is the equivalence relation on $Y$ generated by pairs $(f(x), g(x))$ for $x \in X$, and $q : Y \to Q$ maps each $y \in Y$ to its equivalence class. See 3.131.

In Listing 17 we give a Haskell implementation of coequalizers in the simplicial category $\Delta$. The type Function of functions was defined in Listing 11. Note that the Haskell list comprehension syntax allows us to give an implementation that is almost identical to the mathematical definition.
9.1. HASKELL IMPLEMENTATION

A coequalizer of two functions \( f : X \rightarrow Y \) and \( g : X \rightarrow Y \) is a function \( q : Y \rightarrow Q \) that satisfies a universal mapping property. One solution has \( Q = Y/\sim \) where \( \sim \) is the equivalence relation on \( Y \) generated pairs \( (f(x), g(x)) \) where \( x \) ranges over \( X \).

Listing 17: Coequalizers implemented in Haskell

```haskell
-- A coequalizer of two functions f : X \rightarrow Y and g : X \rightarrow Y is a function
-- q : Y \rightarrow Q that satisfies a universal mapping property. One solution
-- has Q = Y/\sim where \( \sim \) is the equivalence relation on Y generated pairs
-- (f(x), g(x)) where x ranges over x.
coequalizer :: Function \(\rightarrow\) Function \(\rightarrow\) Function
coequalizer f g = (q, y')
    where y = snd f
          pairs = zip (fst f) (fst g) -- edges in a graph with y as vertex set
          startIndices = [0..(y-1)]
          q = shuffle $ foldl convertByIndex startIndices pairs
          y' = length $ nub q

-- Change the value y to x if x == x', otherwise, return y. In C, the
-- swap function could be programmed as swap(x,x')(y) = (x == x') ? x : y
swap :: (Integral a) \(\rightarrow\) a \(\rightarrow\) a \(\rightarrow\) a
swap x x' y
    | y == x' = x
    | otherwise = y

-- If x <= x', then convert \{x0, ..., xn\} (x, x') changes each occurrence
-- of x in the list to x'. Otherwise, it converts each occurrence of x' to x.
convert :: (Integral a) \(\Rightarrow\) [a] \(\rightarrow\) (a, a) \(\rightarrow\) [a]
convert xs (x,x')
    | x <= x' = map (swap x x') xs
    | otherwise = map (swap x' x ) xs

-- Similar to convert but the input pair (i,j) refers to indices in the list.
convertByIndex :: [Int] \(\rightarrow\) (Int, Int) \(\rightarrow\) [Int]
convertByIndex xs (i,j) = convert xs (x,x')
    where x = xs!!i
          x' = xs!!j

-- Ensure that the integers occurring in a list would be \{0, ..., n\} after sorting.
-- For example, \{0, 1, 3\} would be converted to \{0, 1, 2\}.
shuffle :: [Int] \(\rightarrow\) [Int]
shuffle xs = foldl convert xs pairs
    where g = sort $ nub xs
          f = [0..(length g - 1)]
          pairs = zip f g

If we define the following in a Haskell script

\[
\begin{align*}
f &:: Function \\
f &= ([0,1,2],4) \\
g &:: Function \\
g &= ([1,1,3],4)
\end{align*}
\]

then we can use ghci to test our pullback implementation:

\[
\begin{align*}
\text{> coequalizer f g} \\
& ([0,0,1,1],2) \\
\text{> compose q f} \\
& ([0,0,1],2)
\end{align*}
\]
> compose q g
  ([0,0,1],2)

Since \( f(0) = 0 \sim 1 = g(0) \) and \( f(2) = 2 \sim g(2) = 3 \), the set \( Y \) is partitioned into two equivalence classes, \( \{0,1\} \) and \( \{2,3\} \). The coequalizer maps members of \( Y \) to their equivalence classes.

**9.13 Todd-Coxeter Procedure.** Recall that the Todd-Coxeter Procedure is a special case of the Carmody-Walters Algorithm for computing left cosets. We designed our Haskell implementation of the algorithm to exploit the lazy evaluation mechanism of Haskell in order to explore the on-the-fly circuit generation feature of the Quipper language.

**9.2 Quipper Implementation**

Quipper is an programming language for quantum computation that is under development by a team led by Scott Alexander (Applied Communication Sciences) and includes Alexander S. Green (Dalhousie University), Peter LeFanu Lumsdaine (Dalhousie University), Peter Selinger (Dalhousie University) and Benoît Valiron (University of Pennsylvania). Quipper is embedded in the Haskell programming language.

Once we had begun discovering quantum algorithms for computing Kan extensions, we began development of the Java implementations of these algorithms discussed in Chapter 8. Our implementation was rather literal, meaning we stored and manipulated complex matrices and vectors to represent and calculate with quantum circuits. One consequence of this approach was, of course, the fact that the circuits we implemented had to contain only a dozen or so wires. This significantly limited the sizes of the circuits with which we could work.

Researching and experimenting with Quipper, however, was liberating. First, Quipper maintains a symbolic representation and so can manage circuits with hundreds to thousands of wires and gates. This is not possible with a literal matrix representation. Second, the Quipper distribution includes a large collection of built-in gates and valuable libraries such as quantum integers and arithmetic, random-access memory, and the quantum Fourier transform. Third, Quipper integrates powerful abstraction mechanisms (some of which are adapted from the Haskell programming language). These allow the programmer to think in terms of programming idioms rather than individual quantum wires and gates.

**9.21 Quantum Circuits.** In the following subsections we discuss implementations of Kan extension algorithms using Quipper. One approach to building quantum circuits in Quipper is to convert the Haskell implementations that we discussed earlier in this chapter to reversible circuits using Quipper’s `build_circuit` mechanism. This generates a circuit building function of type

```haskell
template_f :: (QNum a) => Circ (a -> Circ a)
```

which one then applies to arguments of the `QNum` type class. During the period on which we were researching this approach, the `build_circuit` mechanism was not currently working. According to the Quipper team, this was due to (1) the fact that some of the required template functions that we required in the `Libraries.Arith` had not yet been implemented and (2) errors detected by the type-checker which may be due to recent changes in the `QCData` implementation.
9.211 Products. In 7.11 we discussed quantum circuits for computing products. In the simplicial category $\Delta$, products are computed by multiplication of natural numbers. In 7.111 we discussed known quantum algorithms for multiplication. The Quipper library Libraries.Arith implements multiplication of fixed-size integers in the function

$q\_mult:: qa \to qa \to \text{Circ}(qa, qa, qa)$

of QData type class. The algorithms is $O(n^2)$ where $n$ is the number of bits required to store the inputs.

9.212 Equalizers. In 7.12 we discussed quantum algorithms for computing equalizers. and in 7.133 we proved that quantum computers can not improve upon classical (probabilistic) complexity of exact equalizer calculations. There are two approaches to using Quipper to implement quantum circuits for computing equalizers. The first is to use the Haskell implementation of equalizers in Appendix C and use the Quipper transformer to generate a reversible circuit. The second is to use the quantum counting algorithm to estimate the size of the equalizer, then use Grover’s Algorithm to find them discussed in 7.121. Grover’s Algorithm is implemented in the Quipper implementation of the Triangle Finding Algorithm. The quantum Fourier transform and its inverse are implemented in Libraries.QFT. These give the components of the quantum counting circuit shown in Section 6.3 of [112].

9.213 Coproducts. We Implemented a quantum coproduct algorithm directly in Quipper and experimented with Quipper output formats for circuits (see Figure 9.1). Our Quipper implementation of this algorithm is shown in Figure 18 on page 125. Note that our implementation, based on [51] as discussed in 7.21, is significantly smaller than the reversible circuit that was automatically generated by Quipper and which is described in the Quipper documentation. The Quipper library Libraries.Arith implements addition of fixed-size integers in the function

$q\_add:: qa \to qa \to \text{Circ}(qa, qa, qa)$

of QData type class. The algorithms is $O(n)$ where $n$ is the number of bits required to store the inputs.

![Figure 9.1: Quantum coproduct circuit implemented in Quipper](image)

9.22 Language Features. There are several features of Quipper that we found useful in the development and understanding of quantum algorithms. The availability of a large number of gates and libraries is highly-valuable as are the abilities to include comments and labels in circuit diagrams and to export circuit diagrams to Postscript files. The fact that Quipper can manage enormous quantum circuits is also valuable.
Quipper integrates advanced features such as on-the-fly circuit generation, dynamic lifting, a circuit transformation interface and boxed subroutines. The most important feature for our work was on-the-fly circuit generation which allows Quipper to operate on descriptions of infinite circuits using a Haskell-like lazy evaluation. We exploited lazy evaluation in our Haskell implementation of the Todd-Coxeter Procedure. In this procedure, the number of cosets is unknown a priori. In fact, it is not known in advance whether or not there is a finite number of cosets. Our Haskell implementation initializes the list of cosets using an infinite list of \texttt{Nothing} entries. These entries may be replaced by numeric values if the progress of the algorithm demands it. See 9.13. Lazy evaluation and on-the-fly circuit generation allow Haskell and Quipper to work with infinite structures such as these.

We found it convenient that Quipper is embedded in an existing programming language, namely Haskell, that is freely-available (see Appendix D) across computing platforms such as Linux, OSX and MS Windows. Although we were new to Haskell programming, the embedded nature of Quipper allowed us to begin experimenting fairly quickly.

### 9.3 QuaFL

According to the QuaFL software guide, “QuaFL is the TORQUE programming language for describing quantum algorithms, from quantum operation on quantum data, to classical operations on quantum data, and to classical operation on classical outcomes of measurements on quantum data.” Like Quipper, QuaFL is a functional programming language. Unlike Quipper, however, QuaFL is not embedded in another language. The syntax of QuaFL programs can be analyzed through a plugin to the Eclipse (www.eclipse.org) integrated development environment (IDE). A design feature of QuaFL is to enable a compiler to generate circuits that minimize the quantum resources required to execute algorithms.

Below we discuss QuaFL data types and abstraction mechanisms that support implementing quantum algorithms for computing Kan extensions.

### 9.31 Basic Data Types

This discussion is based on the QuaFL Programming Guide that was provided to us by Rich Lazarus and Marcus Silva of the BBN TORQUE Team.

QuaFL supports basic data types including \texttt{int} and \texttt{double} types for classical computation, \texttt{rational}, \texttt{bool}, \texttt{bit} and \texttt{uint}[N] types. The latter are binary vector of fixed length N. Our implementation of quantum algorithms in Java, discussed in Chapter 8, built circuits from registers of binary values. The QuaFL \texttt{uint}[N] type supports this construction directly.

### 9.32 Abstraction Mechanisms

QuaFL supports several constructions of more complex types from the basic types. These mechanisms include constructions of both quantum and classical function types, ordered tuples of types, finite lists and fixed-size arrays. Our Haskell implementations of terminal objects 9.111, products 9.112, equalizers 9.113, pullbacks 9.114, initial objects 9.121, coproducts 9.122 and coequalizers 9.123 made use of a subset of the mechanisms available in QuaFL.

Our Haskell implementation of the Todd-Coxeter Procedure, however, employed infinite lists by exploiting the lazy evaluation mechanism of Haskell. We implemented the algorithm in this way because the size of the solution (i.e., the number of cosets) is not known a priori and, as shown in Table 4.1 on page 70, the space requirements during the procedure may be quite high compared to the size of the solution set. The procedure could, of course be implemented with finite length lists as we did in our Java implementation.
of the Carmody-Walters Algorithm.

This raises a question: What quantum resources would be required to execute these algorithms? The QuaFL documentation discusses a two-step process of turning code into a quantum computation: compilation and synthesis. The latter involves determination of the quantum circuits on which the computational process is to be executed. How will synthesis of quantum circuits from code that exploits lazy evaluation differ from that involving fixed-size data structures?

---

9.3. QUAFL

import Quipper

maj : (Qubit, Qubit, Qubit) -> Circ (Qubit, Qubit, Qubit)
maj (q1, q2, q3) = do
q2 <- qnot q2 'controlled' q3
q1 <- qnot q1 'controlled' q3
q3 <- qnot q3 'controlled' [q1, q2]
return (q1, q2, q3)

uma : (Qubit, Qubit, Qubit) -> Circ (Qubit, Qubit, Qubit)
uma (q1, q2, q3) = do
q3 <- qnot q3 'controlled' [q1, q2]
q1 <- qnot q1 'controlled' q3
q2 <- qnot q2 'controlled' q1
return (q1, q2, q3)

six_bit_add : (Bool, Bool, Bool, Bool, Bool, Bool, Bool, Bool, Bool) ->
Circ (Qubit, Qubit, Qubit, Qubit, Qubit, Qubit, Qubit, Qubit, Qubit, Qubit, Qubit, Qubit)
six_bit_add (c0, b0, b1, b2, b3, b4, b5, a0, a1, a2, a3, a4, a5, z) = do
q0 <- qinit False
(q1, q2, q3, q4, q5, q6) <- qinit (b0, a0, b1, a1, b2, a2)
(q7, q8, q9, q10, q11, q12) <- qinit (b3, a3, b4, a4, b5, a5)
q13 <- qinit False
comment_with_label "" [q0, q1, q2, q3, q4, q5, q6, q7, q8, q9, q10, q11, q12, q13] ["c0", "b0", "a0", "b1" ", "a1", "b2", "a2", "b3", "a3", "b4", "a4", "b5", "a5", "z"]
(q0, q1, q2) <- maj (q0, q1, q2)
(q2, q3, q4) <- maj (q2, q3, q4)
(q4, q5, q6) <- maj (q4, q5, q6)
(q6, q7, q8) <- maj (q6, q7, q8)
(q8, q9, q10) <- maj (q8, q9, q10)
(q10, q11, q12) <- maj (q10, q11, q12)
q13 <- cnot q13 'controlled' q12
(q10, q11, q12) <- uma (q10, q11, q12)
(q8, q9, q10) <- uma (q8, q9, q10)
(q6, q7, q8) <- uma (q6, q7, q8)
(q4, q5, q6) <- uma (q4, q5, q6)
(q2, q3, q4) <- uma (q2, q3, q4)
(q0, q1, q2) <- maj (q0, q1, q2)
comment_with_label "" [q0, q1, q2, q3, q4, q5, q6, q7, q8, q9, q10, q11, q12, q13] ["0", "s0", "a0", "s1", ", "a1", "s2", "a2", "s3", "a3", "s4", "a4", "s5", "a5", "z+s6"]
return (q0, q1, q2, q3, q4, q5, q6, q7, q8, q9, q10, q11, q12, q13)

---

Listing 18: Coproduct circuit implemented in Quipper
Chapter 10

Kan Liftings

10.1 Kan Extensions

For comparative reasons, let’s just remember the definitions of Kan extensions. Let $\mathcal{M}$, $\mathcal{C}$ and $\mathcal{A}$ be categories. A functor $K : \mathcal{M} \to \mathcal{C}$ induces

$$- \circ K : \mathcal{A}^{\mathcal{C}} \to \mathcal{A}^{\mathcal{M}}.$$ 

Left and right adjoints of this functor, if they exist, are called left and right Kan extensions:

$$\begin{array}{c}
\begin{array}{ccc}
\mathcal{A}^{\mathcal{C}} & \xrightarrow{\perp} & \mathcal{A}^{\mathcal{M}} \\
\downarrow K & \downarrow \downarrow & \\
\mathcal{M} & \xrightarrow{\eta} & \mathcal{A}
\end{array}
\end{array}$$

The following diagrams will be helpful:

In detail, for $F \in \mathcal{A}^{\mathcal{C}}$ and $G \in \mathcal{A}^{\mathcal{M}}$ we have

$$\mathcal{C}(\text{Lan}_K(G), F) \cong \mathcal{M}(G, F \circ K)$$

and

$$\mathcal{C}(F, \text{Ran}_K(G)) \cong \mathcal{M}(F \circ K, G).$$

In terms of universal properties this says that there exists

$$\eta : G \implies \text{Lan}_K(G) \circ K$$

127
which is universal from $G$ to $- \circ K$. This means for any other $L : \mathcal{C} \to \mathbb{A}$ and $\eta' : G \Rightarrow L \circ K$, there is a unique natural transformation $\tau : L \Rightarrow \text{Lan}_K(G)$ such that

\[
\begin{array}{ccc}
G & \xrightarrow{\eta} & \text{Lan}_K(G) \circ K \\
\downarrow & & \downarrow \tau \\
L \circ K & \xrightarrow{(\tau \circ K)} & \text{Lan}_K(G) \\
\end{array}
\]

\[\eta' = (\tau \circ K) \cdot \eta\]

For the right Kan extension, this means there exists

\[\varepsilon : \text{Ran}_K(G) \circ K \Rightarrow G\]

which is universal from $- \circ K$ to $G$. This means that for any other $R$ and $\varepsilon' : R \circ K \Rightarrow G$ there is a unique natural transformation $\sigma : R \Rightarrow \text{Ran}_K(G)$ such that

\[
\begin{array}{ccc}
\text{Ran}_K(G) & \xrightarrow{\sigma} & R \\
\downarrow & \downarrow (\sigma \circ K) & \downarrow \varepsilon \\
R \circ K & \xrightarrow{\varepsilon'} & G \\
\end{array}
\]

\[\varepsilon' = \varepsilon \cdot (\sigma \circ K)\]
and

$$\mathbb{C}^M(F, \operatorname{Rif}_P(G)) \cong \mathbb{A}^M(P \circ F, G).$$

The following diagrams will be helpful:

In terms of universal properties this says that there exists

$$\eta : G \Longrightarrow P \circ \operatorname{Lif}_P(G)$$

is universal from $G$ to $P \circ -$. This means for any other $L : M \to C$ and $\eta' : G \Longrightarrow P \circ L$, there is a unique natural transformation $\tau : L \Longrightarrow \operatorname{Lif}_P(G)$ such that

$$\eta' = (P \circ \tau) \cdot \eta$$

For the right Kan lifting, this means that there exists

$$\varepsilon : P \circ \operatorname{Rif}_P(G) \Longrightarrow G$$

which is universal from $P \circ -$ to $G$. This means that for any other $R$ and $\varepsilon' : P \circ R \Longrightarrow G$ there is a unique natural transformation $\sigma : R \Longrightarrow \operatorname{Rif}_P(G)$ such that

That is

$$\varepsilon' = \varepsilon \cdot (P \circ \sigma).$$
10.21 Kan Liftings as Limits and Colimits. For functors $G : \mathcal{M} \rightarrow \mathcal{A}$, and $P : \mathcal{C} \rightarrow \mathcal{A}$, and object $m$ in $\mathcal{M}$ consider the arrow category $(G(m) \downarrow P)$ which consists of arrows

$$f : G(m) \rightarrow P(c)$$

in $\mathcal{A}$ for some $c$ in $\mathcal{C}$. The morphisms of this category consists of commuting triangles

$$\begin{array}{c}
G(m) \\
\downarrow f \\
P(c) \xrightarrow{P(g)} P(c') \\
\downarrow f_1 \\
\end{array}$$

for some $g : c \rightarrow c'$ in $\mathcal{C}$. This category has an obvious forgetful functor

$$Q_m : (G(m) \downarrow P) \rightarrow \mathcal{C}$$

which forgets the arrows $f$ and $f_1$ and just sends the above diagram to the $g : c \rightarrow c'$ arrow in $\mathcal{C}$. If the left Kan lifting exists, it can be calculated on the object $m \in \mathcal{M}$ as

$$\text{Lif}_P(G)(m) = \text{Lim}(Q_m : (G(m) \downarrow P) \rightarrow \mathcal{C}).$$

On morphisms this functor is defined as follows. Let $h : m \rightarrow m'$ be an arrow in $\mathcal{M}$. Then we have the following diagram:

$$\begin{array}{c}
G(m) \\
\downarrow f \\
P(c) \xrightarrow{P(g)} P(c_1) \\
\downarrow f_1 \\
G(h) \\
\downarrow f_2 \\
P(c_2) \xrightarrow{P(g')} P(c_3) \\
\downarrow f_3 \\
G(m') \\
\end{array}$$

Notice that $f_2 \circ G(h) : G(m) \rightarrow P(c_2)$ is an arrow in $\mathcal{A}$ and hence an element in $(G(m) \downarrow P)$. So $c_2$ will be in the image of the forgetful functor $Q_m$. By the universal property of $\text{Lif}_P(G)(m') = \text{Lim}(Q_{m'})$ there is an induced

$$\text{Lif}_P(G)(m) = \text{Lim}(Q_m)$$

which we shall call $\text{Lif}_P(G)(h)$.

When $P$ and $G$ are fixed we shorten $\text{Lif}_P(G)$ to $\text{Lif}$.

Just because $\text{Lif}$ exists as a functor does not mean that it satisfies the universal properties. Such universal properties only occur when we assume that $P$ preserves limits, that is, $P$ is continuous. This is in contrast to the case of the Kan extension where the existence only depends on the two categories related, not to the functor in which we are taking the extension.
First let us define the components of the natural transformation \( \eta \). Let \( m \) be an element of \( \mathbb{M} \). \( G(m) \) is an element of \( \mathbb{A} \). Let \( f : G(m) \to P(c) \) for some \( c \) in \( \mathbb{C} \). There is going to be a

\[
\lambda_f : \text{Lif}(m) \to c
\]

in \( \mathbb{C} \). The image of this function in \( \mathbb{A} \) is

\[
P(\lambda_f) : P(\text{Lif}(m)) \to P(c).
\]

Since we assume that \( P \) preserves limits, \( P(\text{Lif}(m)) \) is a limit and of all the \( P(c) \)'s. Since there are \( f : G(m) \to P(c) \)'s, there is an induced

\[
\eta_m : G(m) \to P\text{Lif}(m)
\]

which are the components of the \( \eta \). This satisfies

\[
f = P(\lambda_f) \circ \eta_m.
\]

Now for the naturality of \( \eta \). Let \( h : m \to m' \) be an arrow in \( \mathbb{M} \). This induces the following diagram.

Since there is a map \( f_3 \circ G(h) \) there is an induced diagonal map \( d : G(m) \to P\text{Lif}(m') \). This satisfies

\[
f_3 \circ G(h) = P(\lambda_{f_3}) \circ d
\]

Lif satisfies the universal property.

There are different levels of \( P \) being an “onto” function. From stronger to weaker we have

- \( P \) is surjective if for every \( a \in \mathbb{A} \) there exists a \( c \in \mathbb{C} \) such that \( P(c) = a \).
- \( P \) is surjective on \( G \) if for every \( m \) in \( \mathbb{M} \) there exists a \( c \in \mathbb{C} \) such that \( G(m) = P(c) \).
- \( P \) is left weakly surjective on \( G \) if for every \( m \) in \( \mathbb{M} \) there exists a \( c \) in \( \mathbb{C} \) and a map \( f : G(m) \to P(c) \) in \( \mathbb{A} \) such that for any other \( f' : G(m) \to P(c') \) there is a unique \( \alpha : P(c) \to P(c') \) having \( \alpha \circ f = f' \). This essentially says that every \( G(m) \) is “covered” and is “covered” by a smallest element in the image of \( P \).

We state the next theorem with the weakest hypothesis.

**Theorem 1** Let \( G : \mathbb{M} \to \mathbb{A} \) and \( P : \mathbb{C} \to \mathbb{A} \) be weakly surjective on \( G \). Then \( \eta \) is the identity, i.e.

\[
G \cong P \circ \text{Lif}_P(G).
\]
The right Kan lifting is done similarly with the category $P \downarrow G(m)$ and a similar forgetful functor $Q'_m : P \downarrow G(m) \to \mathbb{C}$.

$$\text{Rif}_P(G)(m) = \text{Colim}(Q'_m : (P \downarrow G(m)) \to \mathbb{C})$$

It does not take much to show that $\text{Rif}_P(G)$ is a well-defined functor from $\mathbb{M}$ to $\mathbb{C}$.

One of the most interesting aspects of Kan liftings is that it has been basically ignored in the literature. There does not seem to be any major papers on the subject. This is strange since it seems most natural.

Liftings as Ends and Coends. First we remember the definition of Kan extensions in terms of ends and coends

$$\text{Lan}_K(m) = \int^c \mathbb{C}(Km, c) \cdot G(m)$$

$$\text{Ran}_K(m) = \int_c G(m)^{\mathbb{C}(c,Km)}$$

Consider $(G(m) \downarrow P)$ and $(G(m') \downarrow P)$. Notice that $h : m \to m'$ induces

$$h^* : (G(m') \downarrow P) \to (G(m) \downarrow P)$$

and that it satisfies

$$Q'_m = Q_m \circ h^*.$$

In other words, the arrows in $\mathbb{M}$ co-act on the functor

$$Q_{(-)}(G(\cdot) \downarrow P).$$

$$\text{Lif}_K(m) = \int^c \mathbb{C}(Km, c) \cdot Q(G(\cdot) \downarrow P) \cdot$$

$$\text{Rif}_K(m) = \int_c G(m)^{\mathbb{C}(c,Km)}$$

We note that the following properties of liftings

$$P \circ P' \quad G \circ G'$$

are proved with the Fubini property of ends and coends.

10.22 Examples of Kan lifting. A primary example of the uses of Kan liftings is in the area of algebraic theories. Let $T$ and $T'$ be algebraic theories a la Lawvere’s thesis. Let $\mathbb{C}$ be a category with products. Then the category of product preserving functors from $T$ to $\mathbb{C}$ is the categories of algebras of $T$ in $\mathbb{C}$. This category is denoted as

$$\text{Alg}(T, \mathbb{C}) \subseteq \mathbb{C}_T.$$

If $F : T \to T'$ is a theory morphism, i.e., $F$ preserves products and $F(1) = 1$ then there is an induced functor

$$F^* : \text{Alg}(T', \mathbb{C}) \to \text{Alg}(T, \mathbb{C})$$
via pre-composition. Left and right adjoints to $F^*$ are the left and right Kan extensions along $F$. So if $a : T \to \mathbb{C}$ is an algebra for $T$ in $\mathbb{C}$ then the Kan extension

\[
\begin{array}{ccc}
T' & \xrightarrow{\text{Kan extension}} & \mathbb{C} \\
\downarrow F & & \downarrow a \\
T & &
\end{array}
\]

is an algebra for $T'$ in $\mathbb{C}$. This is a classical theorem of algebraic theories.

Now for the more interesting case of an application of Kan lifting. Let $\mathbb{C}'$ be another category with products. Then we have

\[
\text{Alg}(T, \mathbb{C}') \subseteq \mathbb{C}'^T.
\]

For a product preserving functor $G : \mathbb{C} \to \mathbb{C}'$ there is an induced

\[
G_* : \text{Alg}(T, \mathbb{C}) \to \text{Alg}(T, \mathbb{C}')
\]

via post-composition. Left and right adjoints of this functor are given by Kan liftings as follows. Given an algebra $a : T \to \mathbb{C}'$ we have the a Kan lifting

\[
\begin{array}{ccc}
T & \xrightarrow{\text{Kan lifting}} & \mathbb{C} \\
\downarrow a & & \downarrow G \\
\mathbb{C}' & &
\end{array}
\]

which gives an algebra of $T$ in $\mathbb{C}$. For example, Let $\mathbb{C} = \textbf{Top}$ and $\mathbb{C}' = \textbf{Set}$ be the categories of topological spaces and Sets. Let $G$ be the forgetful function from $\textbf{Top}$ to $\textbf{Set}$. Let $T$ be the theory of groups. Then

\[
G_* : \text{Alg}(T, \mathbb{C}) = \textbf{TopGrps} \to \text{Alg}(T, \mathbb{C}') = \textbf{Grps}.
\]

The Kan lifting gives the free topological group over a group.

Kan liftings can also be found in the area of weighted limits and colimits as well as relative adjoint functors. We leave this to the reader to discover.
Chapter 11

Abstract Homotopy Theory

In order to understand Kan homotopy extensions homotopy, we have to go back to its origins: abstract homotopy theory. There are many different categories that describe topological spaces. There is, for example, the category of topological spaces, simplicial sets, simplicial complexes, and chain complexes. While all these categories are different they are trying to describe the same thing. Abstract homotopy theory was formulated to show how they are the same. In 1967, Daniel Quillen showed one how to talk about homotopy theory from a categorical point of view ([115] see also [62, 81]). A category $\mathcal{C}$ has a Quillen model category structure if there are three classes of morphisms in $\mathcal{C}$ called weak equivalences, fibrations and cofibrations. The category $\mathcal{C}$ and these classes of maps must satisfy axioms (see below). Once one has such a structure, one can go on to create the analogies of path spaces, mapping cylinders, Puppe sequences and all other important tools of homotopy theory. Given a Quillen model structure, one formally inverts the weak equivalences, that is, makes the weak equivalences into isomorphisms, in order to construct the homotopy category, $Ho(\mathcal{C})$, and a functor $\gamma: \mathcal{C} \to Ho(\mathcal{C})$. We can then ask if homotopy categories are equivalent. It turns out that all the different ways of describing topological spaces have equivalent homotopy categories.

Over the past 45 years, researchers have described many other categories that have Quillen model categories structures such as DG-Algebras, Chain complexes of algebraic structure, the category of categories, higher categories, operads, 2-theories [135] etc. These many different structures have shown that homotopy theory has a fundamental role in mathematics, theoretical computer science and theoretical physics. Our aim here is to show that the notions of homotopy can have an important role in quantum computing.

First let us describe the notions of a Quillen model category structure. Let $\mathcal{C}$ be a small category with three classes of maps:

- Weak Equivalences (a superset of isomorphisms) (To be thought of as maps between objects that are “essentially” the same.)
- Fibrations (To be thought of as “nice” onto maps.)
- Cofibrations (To be thought of as “nice” into maps.)

Furthermore the category and these subclasses must satisfy the following axioms:

1. **Limits and Colimits.** $\mathcal{C}$ has all finite limits and colimits. $\square$

2. **Two out of Three.** If $f, g$ and $g \circ f$ are morphisms in $\mathcal{C}$ and any two of them are weak equivalences, then so is the third. $\square$
3. **Retracts.** If $f$ is a retract of $g$, in $\mathbb{C}$ and if $g$ is a weak equivalence (resp. fibration, cofibration) then $f$ is also a weak equivalence (resp. fibration, cofibration). □

4. **Lifting Axioms.** Consider the following commutative diagram:

\[
\begin{array}{ccc}
A & \rightarrow & B \\
\downarrow \text{Cof} & & \downarrow \text{Fib} \\
C & \rightarrow & D
\end{array}
\]

If the left or right vertical map is also a weak equivalence, then there exists a lifting $H$ making the two triangles commute. □

5. **Factorization Axioms.** Every morphism $f : A \rightarrow B$ can be factored as a weak equivalence followed by a fibration; and also a cofibration followed by a weak equivalence and a fibration as in the following.

\[
\begin{array}{ccc}
C & \rightarrow & C \\
\downarrow \text{W.E.} \cap \text{Cof} & & \downarrow \text{W.E.} \cap \text{Fib} \\
A & \rightarrow & B \\
\downarrow \text{Cof} & & \downarrow \text{W.E.} \cap \text{Fib} \\
D
\end{array}
\]

□

With this structure, we invert the weak equivalences to construct $Ho(\mathbb{C})$, the homotopy category of $\mathbb{C}$, and a functor $\gamma : \mathbb{C} \rightarrow Ho(\mathbb{C})$. $\gamma$ is universal with respect to functors that invert the weak equivalences. That means if there is any other category $\mathbb{D}$ and functor $\alpha : \mathbb{C} \rightarrow \mathbb{D}$ such that $\alpha$ takes all weak equivalences to isomorphisms in $\mathbb{D}$ then there is a unique $\beta : Ho(\mathbb{C}) \rightarrow \mathbb{D}$ such that

\[
\begin{array}{ccc}
\mathbb{C} & \rightarrow & \mathbb{D} \\
\gamma & \downarrow & \alpha \\
Ho(\mathbb{C}) & \rightarrow & D
\end{array}
\]

commutes. In other words, $\gamma$ inverts only the weak equivalences and $Ho(\mathbb{C})$ is the best fitting.

A few remarks that we will need have to be made:

- First given the weak equivalences and the fibrations, one can determine the cofibrations. Similarly, given the weak equivalences and the cofibrations, one can determine the fibrations.
11.1. HOMOTOPY KAN EXTENSIONS AND LIFTINGS

• Another interesting aspect of this definition is that the axioms are totally symmetric. In other words, if \( C \) has a Quillen model category structure, then inverting the fibrations and the cofibrations gives \( C^{op} \) also has a Quillen model category structure.

• Notice that the fibrations and the cofibrations are not needed for the definitions of \( Ho(C) \). Only the weak equivalences are used for that construction. The fibrations and cofibrations are however needed for other constructions like homotopy limits and colimits.

11.1 Homotopy Kan Extensions and Liftings

We have the following Proposition from [115].

**Proposition 1** Let \( C \) and \( C' \) be model categories and let

\[
\begin{array}{c}
C \\
\downarrow^L
\end{array}
\quad \quad
\begin{array}{c}
\downarrow^R
C'
\end{array}
\]

be a pair of adjoint functors. Suppose \( L \) preserves cofibrations and \( R \) preserves fibrations. Then the left Kan extension \( \text{Lan}_{\gamma'}(\gamma \circ L) \) and the right Kan extension \( \text{Ran}_{\gamma'}(\gamma \circ R) \) exists and are adjoint.

If \( C \) has a model category structure and \( D \) is a category then \( C^D \) has at least two possible model category structures.

• The projective structure has the weak equivalences and fibrations are the natural transformations that are objectwise such morphisms in \( C \). The cofibrations are generated from the other two. And

• The injective structure has the weak equivalences and cofibrations are the natural transformations that are objectwise such morphisms in \( C \). The fibrations are generated from the other two.

These two are also called “gross” and “petit” in the literature.

It is important to realize that \( Ho(C^D) \) is not the same as \( Ho(C)^D \). In fact, a large part of this theory is based on how different they are. There is, however, an induced functor

\[
I : Ho(C^D) \longrightarrow Ho(C)^D.
\]

What this means is that the homotopy Kan extension is NOT the Kan extension in the homotopy category. Rather it is induced by the Kan extension in the Quillen model category structures.
For our concern, we are interested where the Quillen model category structure on the category of sets which is the category where our Kan extensions and liftings happen. The category of set does not have a non-trivial Quillen model category structure. A trivial one is where the weak equivalences are only the isomorphisms. In this case the homotopy category is nothing more than the original category. Rather we shall use a category used in [99]. The category is PartSet of partitioned sets. The objects are pairs $(A, \sim_A)$ where $A$ is a set and $\sim_A$ is an equivalence relation on $A$. Morphisms $f : (A, \sim_A) \to (B, \sim_B)$ are set functions that preserve the relation. That is, If $a \sim_A a'$ then $f(a) \sim_B f(a')$. The Quillen model structure is given as follows:

- Weak equivalences are maps that induce a bijection of quotient sets.
- Fibrations are maps that map onto classes of the target set.
- Cofibrations are injective set maps.

See [99] for the details which show that this is an Quillen model category. In fact, there are very nice properties of this model category structure: it is locally finitely presentable, cofibrantly generated, and not cellular.

Let $K : M \to C$ be an arbitrary function then we have induced

$$K^* : \text{PartSet}^C \to \text{PartSet}^M$$

These two categories of diagrams have Quillen model category structures. Consider their induced homotopy category. This gives us the following diagram:

$$
\begin{array}{ccc}
\text{PartSet}^C & \xrightarrow{K^*} & \text{PartSet}^M \\
\downarrow \text{HoLan}_K & & \downarrow \text{HoRan}_K \\
\text{Ho}(\text{PartSet}^C) & \xleftarrow{\text{Ho}(K^*)} & \text{Ho}(\text{PartSet}^M)
\end{array}
$$

$\text{HoLan}$ and $\text{HoRan}$ are the homotopy Kan left and right extensions along $K$.

One can also formulate the notions of left and right homotopy Kan liftings.

It is not hard to formulate a classical algorithm a la [36] to calculate a homotopy Kan extension. The input would have to take into account not only the set but the equivalence relation on the set. Basically the algorithm sets up the same tables as in regular Kan extensions. However when there are “coincidences” in the tables, i.e., values that are different when they are supposed to be the same, we have a different procedure. In the classical algorithm we have to change the values. In the algorithm for the homotopy Kan extension, we might be able to leave the different values alone. If the two values are in the same equivalence class, then they can remain the same. Thus an algorithm for the homotopy Kan extension is a slight modification of the classical algorithm.

What possible uses can there be in the area of quantum algorithms? One of the major problems in quantum computing is keeping data away from the environment. This leaves open a major work in error-correcting codes and fault-tolerance type structure. Imagine a set of data where every piece of data can
be a whole range of data. This is essentially an equivalence relation on a set. As we have seen, many quantum algorithms can be viewed as Kan extensions and Kan liftings on data. We are interested in such Kan phenomena where the data is given with a range of values.

11.2 Cat as a Closed Model Category

Thomason defines a (closed) model structure on $\mathbf{Cat}$, the category of small categories from the structure on $\hat{\Delta}$, the category of simplicial sets, via a pair of adjoint functors:

$\hat{\Delta} \xrightarrow{\mathcal{F}} \mathbf{Cat} \xleftarrow{\mathcal{U}}$

where $\mathcal{U} = \text{Ex} \circ \text{Ex} \circ N$ and $\mathcal{F} = c \circ Sd \circ Sd$.

11.21 Historical Background.

Whitehead introduced the category of CW complexes as the appropriate category in which to do homotopy theory. Eilenberg, Mac Lane and Zilber defined the notion of simplicial set in the early 50’s and Kan introduced the necessary conditions to do homotopy theory in this category. The equivalence of these categories played an important role in the development of geometric topology. In the late 60’s, Quillen used the notion of classifying space for a small category and showed the importance of doing homotopy theory in the category of small categories. Latch showed that the category of small categories and the category of simplicial sets were equivalent up to homotopy. [The standard nerve functor $N : \mathbf{Cat} \to \hat{\Delta}$ gives the equivalence.]

— Fritsch and Latch (1979)

Thomason’s work is part of a larger long-term program to develop algebraic topology of small categories.

Many workers believed for a long time that small categories were nearly homotopically trivial. But then Illusie (1972), Lee (1974), and Latch (1977) showed that the small categories $\Lambda X$ and $\Gamma X$ have the homotopy type of $X$, for any simplicial set $X$. From the geometric point of view, the functors

$\Lambda : \hat{\Delta} \to \mathbf{Cat}$ and $\Gamma : \mathbf{Cat} \to \hat{\Delta}$

are not quite satisfactory: $NAX$ and $N\Gamma X$ are both infinite dimensional simplicial sets for every simplicial set $X$, even the trivial simplicial point $\Delta[0]$. Then Thomason conjectured that the construction $c \circ Sd \circ Sd : \hat{\Delta} \to \hat{\Delta}$ would preserve homotopy type.

— Fritsch and Latch (1981)

The genesis of this paper lies in [Thomason’s proof] that the nerve functor carries lax colimits in the category $\mathbf{Cat}$ of small categories to homotopy colimits in the category $\hat{\Delta}$ of simplicial sets, up to homotopy type. I found that the main properties of homotopy limits and colimits ... did not seem to follow from any well-known categorical techniques. At first I thought what was lacking was something in the 2-categorical nature of homotopy theory, but this (apparently) was incorrect. Instead, what is involved is the closed category nature of both
\textbf{Cat} and $\hat{\Delta}$ and the fact that the nerve functor is a closed functor with a left adjoint. The development of the theory of such closed functors has not been in the appropriate direction to yield the required formulas. In providing this development, it turned out that there is a whole aspect of the study of closed categories and closed functors that consists in applying some simple properties of what is called here a “tensor-hom-cotensor” (THC)-situation in more and more complex circumstances.

— Gray (1980)
11.22 Related References. The following is an incomplete, chronological list of references related to Thomason’s Paper.


1963 Mac Lane: Homology. Springer-Verlag New York, Inc.


1971 Mac Lane: Categories for the Working Mathematician. Springer-Verlag, New York-Berlin.


Hoff: “Introduction à L’Homotopie dans Cat.” Esquisses Math. 23


11.23 Closed Model Categories. A (closed) model structure on a category $\mathcal{C}$ is a triple $(W, E, M)$ of classes of morphisms of $\mathcal{C}$ such that:

1. $W$, $E$, and $M$ contain all identities;
2. $E$ and $M$ are closed under compositions;
3. CM2: $W$ is a saturated;
4. CM3: $W$, $E$, and $M$ are retract closed;
5. CM4: A lift exists for any commutative diagram of one of the following two forms:

$$\begin{array}{ccc}
A & \xrightarrow{f} & X \\
\downarrow{i} & & \downarrow{p} \\
B & \xrightarrow{g} & Y
\end{array} \quad \begin{array}{ccc}
A & \xrightarrow{f} & X \\
\downarrow{i} & & \downarrow{p} \\
B & \xrightarrow{g} & Y
\end{array}$$

6. CM5: $(W \cap M, E)$ and $(M, W \cap E)$ are factorization systems for $\mathcal{C}$.

A (closed) model category is a pair $(\mathcal{C}, (W, E, M))$ with $\mathcal{C}$ a finitely complete and finitely cocomplete category and $(W, E, M)$ a (closed) model structure on $\mathcal{C}$.

Definition: A (closed) model category $(\mathcal{C}, (W, E, M))$ is proper if it satisfies the following two axioms of propriety:

P1: If the diagram on the left is a pushout diagram with $i \in M$ and $f \in W$ then $g \in W$:

$$\begin{array}{ccc}
A & \xrightarrow{f \in W} & C \\
\downarrow{i} & & \downarrow{p} \\
B & \xrightarrow{g} & D
\end{array}$$

P2: If the diagram on the right is a pullback diagram with $p \in E$ and $g \in W$ then $f \in W$:

$$\begin{array}{ccc}
A & \xrightarrow{f} & C \\
\downarrow{i} & & \downarrow{p \in E} \\
B & \xrightarrow{g \in W} & D
\end{array}$$

11.231 Our Main Characters.

There is a proper (closed) model structure $(W, E, M)$ on $\hat{\Delta} = \text{Set}^{\Delta^0}$.

There is a nerve functor $N : \text{Cat} \to \hat{\Delta}$ acting on objects by:

$$N(C) = \text{Cat}(i(\_), C) : \Delta^0 \to \text{Set}$$

where $i : \Delta^0 \to \text{Cat}^0$ is inclusion.

Nerve has a left adjoint $c : \hat{\Delta} \to \text{Cat}$.

Let $Sd : \hat{\Delta} \to \hat{\Delta}$ be the subdivision functor. (See Kan: "On C.S.S. Complexes" 1958.)

Then $Sd$ has a right adjoint $Ex : \hat{\Delta} \to \hat{\Delta}$.

The functors:

$$\mathcal{U} = Ex \circ Ex \circ N : \text{Cat} \to \hat{\Delta} \quad \text{and}$$

$$\mathcal{F} = c \circ Sd \circ Sd : \hat{\Delta} \to \text{Cat}$$
are used to lift the (closed) model structure from $\hat{\Delta}$ to one on $\text{Cat}$. Note that $\mathcal{F} \dashv \mathcal{U}$.

We will make use of two sets of morphisms in $\hat{\Delta}$:

1. $\mathcal{H}$, the set of horns. These are inclusions $\Delta[n,k] \to \Delta[n]$ where $\Delta[n] = \Delta(\_ , n) : \Delta^\circ \to \text{Set}$;
2. ii) $\mathcal{H}'$, the canonical boundary inclusions $\hat{\Delta}[n] \to \Delta[n]$.

We need the following facts about these sets:

1. $\mathcal{H} \subset \mathcal{W} \cap \mathcal{M}$ and $\mathcal{F}(\mathcal{H}) \subset \mathcal{W}' \cap \mathcal{M}'$ where $(\mathcal{W}', \mathcal{E}', \mathcal{M}')$ is the (closed) model structure on $\text{Cat}$;
2. ii) $\mathcal{H}' \subset \mathcal{M}$.

11.24 A Closed Model Structure on $\text{Cat}$. From the (closed) model structure $(\mathcal{W}, \mathcal{E}, \mathcal{M})$ on $\hat{\Delta}$ a (closed) model structure $(\mathcal{W}', \mathcal{E}', \mathcal{M}')$ on $\text{Cat}$ is defined as follows: For a functor $f$ between small categories,

$f \in \mathcal{W}'$ if and only if $\mathcal{U}(f) \in \mathcal{W}$;
$f \in \mathcal{E}'$ if and only if $\mathcal{U}(f) \in \mathcal{E}$;
$f \in \mathcal{M}'$ if and only if $f \in \text{LLP}(\mathcal{W}' \cap \mathcal{E}')$.

Moreover, both $(\text{Cat}, (\mathcal{W}', \mathcal{E}', \mathcal{M}'))$ and $(\hat{\Delta}(\mathcal{W}, \mathcal{E}, \mathcal{M}))$ are proper (closed) model categories.

Proof that $(\text{Cat}, (\mathcal{W}', \mathcal{E}', \mathcal{M}'))$ is a (closed) model category is the content of sections 3 and 4 of Thomason’s paper. Propriety and applications are considered in Section 5.

Axioms i), ii), CM2, CM3, CM4(i), and P2 are proved without detailed knowledge of the functors $\mathcal{U}$ and $\mathcal{F}$. That $\mathcal{F} \dashv \mathcal{U}$ enters into the P2 discussion.

Proofs of CM5, CM4(ii), and P1 are more technical. They are discussed in Sections 4 and 5 of the paper.

11.241 $\mathcal{W}', \mathcal{E}'$, and $\mathcal{M}'$ Contain the Identity Morphisms. For any small category $\mathcal{A}$, $\mathcal{U}(1_\mathcal{A}) = 1_{\mathcal{U}(\mathcal{A})} \in \mathcal{W} \cap \mathcal{E}$. So, $1_\mathcal{A} \in \mathcal{W}' \cap \mathcal{E}'$.

Moreover, if the following is commutative in $\text{Cat}$:

$$
\begin{array}{ccc}
A & \xrightarrow{F} & B \\
\mathcal{A} & \xrightarrow{1_\mathcal{A}} & \xrightarrow{P \in \mathcal{W}' \cap \mathcal{E}'} C, \\
\end{array}
$$

then $F$ itself is a lift:

$$
\begin{array}{ccc}
\mathcal{A} & \xrightarrow{F} & B \\
\mathcal{A} & \xrightarrow{1_\mathcal{A}} & \xrightarrow{P \in \mathcal{W}' \cap \mathcal{E}'} C. \\
\end{array}
$$

So $1_\mathcal{A} \in \text{LLP}(\mathcal{W}' \cap \mathcal{E}') = \mathcal{M}'$.

11.242 $\mathcal{E}'$ and $\mathcal{M}'$ are Closed Under Compositions. If $P : \mathcal{A} \to \mathcal{B}$ and $P' : \mathcal{B} \to \mathcal{C}$ are functors between small categories and are also in $\mathcal{E}'$ then $\mathcal{U}(P)$ and $\mathcal{U}(P')$ are in $\mathcal{E}$.

The class $\mathcal{E}$ is closed under compositions so $\mathcal{U}(P') \circ \mathcal{U}(P)$ is in $\mathcal{E}$.

But $\mathcal{U}$ is a functor so $\mathcal{U}(P') \circ \mathcal{U}(P) = \mathcal{U}(P' \circ P)$, hence, $P' \circ P$ is in $\mathcal{E}'$. 
Let $I : \mathcal{A} \to \mathcal{B}$ and $I' : \mathcal{B} \to \mathcal{C}$ be functors between small categories which are also in $M'$ and let the following diagram be commutative in $\textbf{Cat}$:

![Diagram]

That $I$ is in $M'$ implies there is a lift $H$ of the following diagram:

![Diagram]

That $I'$ is in $M'$ implies there is a lift $H'$ of the following diagram:

![Diagram]

This $H'$ is a lift of the original diagram since

$$P \circ H' = G$$

and

$$H' \circ I' \circ I = H \circ I = F.$$

Hence, $I' \circ I \in \text{LLP}(W' \cap E') = M'$.

11.243 Proof of CM2. We must show that $W'$ is saturated. That is, if $F : \mathcal{A} \to \mathcal{B}$ and $G : \mathcal{B} \to \mathcal{C}$ are functors between small categories and any two of $F, G, \text{and } G \circ F$ are in $W'$, so is the third:

![Diagram]

If $F$ and $G$ are in $W'$ then $\mathcal{U}(F)$ and $\mathcal{U}(G)$ are in $W$ by definition of $W'$.

But $W$ is saturated since $(W, E, M)$ is a closed model structure. Hence, $\mathcal{U}(G \circ F)$ is in $W$. This implies by definition of $W'$ that $G \circ F$ is in $W'$.

Proofs of the other two cases are similar. □
Proof of CM3. Suppose \( F : A \to B \) and \( G : X \to Y \) are functors between small categories and that \( F \) is a retract of \( G \).

If \( G \in E' \) then \( U(G) \in E \) by definition of \( E' \). Since \( E \) is closed under retracts, \( U(F) \in E \). Thus, \( F \in E' \) by definition of \( E' \).

The proof for \( G \in W' \) is similar. \( \square \)

To prove CM3 for \( M' \), suppose \( G \in M' \) and that the following is commutative in \( \text{Cat} \):

\[
\begin{array}{ccc}
A & \xrightarrow{S} & A' \\
F & \downarrow & \downarrow \\
B & \xrightarrow{T} & B'.
\end{array}
\]

Then

\[
Q \circ S \circ b = T \circ F \circ b = T \circ d \circ G.
\]

That is, the following is commutative in \( \text{Cat} \) and admits a lift as indicated:

\[
\begin{array}{ccc}
X & \xrightarrow{b} & A & \xrightarrow{S} & A' \\
G \in M' & \downarrow & \downarrow & \downarrow & \downarrow \\
Y & \xrightarrow{d} & B & \xrightarrow{T} & B'.
\end{array}
\]

This implies \( Q \circ L \circ k = T \circ d \circ k = T \) and \( L \circ k \circ F = L \circ G \circ a = S \circ b \circ a = S \). That is, the following is commutative in \( \text{Cat} \) and \( L \circ k \) is a lift for our original diagram:
11.245 Proof of CM4. Given the following commutative diagram in Cat:

\[ \begin{array}{ccc}
A & \xrightarrow{F} & X \\
I \in M' & \downarrow & \downarrow P \in W' \cap E' \\
B & \xrightarrow{G} & Y \\
\end{array} \]

a lift exists since \( M' = LLP(W' \cap E') \).

CM4 is left to be proved. □

11.246 Proof of P2. We must show that if the diagram on the left is a pullback diagram, \( P \in E' \), and \( G \in W' \) then \( F \in W' \):

\[ \begin{array}{ccc}
A & \xrightarrow{F} & C \\
I & \downarrow & \downarrow U(P) \\
B & \xrightarrow{G} & D \\
\end{array} \]

\[ \begin{array}{ccc}
U(A) & \xrightarrow{U(F)} & U(C) \\
\downarrow U(I) & & \downarrow U(G) \\
\downarrow U(P) & & \downarrow U(D) \text{ in Cat} \\
\end{array} \]

in \( \hat{\Delta} \)

If \( P \in E' \) then \( U(P) \in E \) by definition of \( E' \). If \( G \in W' \) then \( U(G) \in W \) by definition of \( W' \). If the diagram in Cat is a pullback diagram then so is the diagram in \( \hat{\Delta} \) since \( U \) has a left adjoint. But \( \hat{\Delta} \) satisfies Property 2 so \( U(F) \in W \). Thus, \( F \in W' \) by definition of \( W' \).

P1 must still be proved. □

11.247 A Construction Used in Proofs of CM5, CM4, and P1 Let \( \mathcal{H} \) be a set of cofibrations in \( \hat{\Delta} \): \( \mathcal{H} \subset M \). By 4.6 of Thomason’s paper, \( F(\mathcal{H}) \subset M' \). For a \( \varpi \in \text{Cat}(S,T) \) construct a factorization \( \varpi = p_{\varpi} \circ u_{\varpi} \) with \( u_{\varpi} \in M' \) as follows.

Let

\[ \mathcal{D}'(\mathcal{H}, \varpi) = \{(F(h), (a, b)) \mid h \in \mathcal{H}, (a, b) \in \text{Ar}_{\text{Cat}}(F(h), \varpi)\}. \]

This is a set. Construe it as a small, discrete category. A member \( D = (F(h), (a, b)) \) of \( \mathcal{D}'(\mathcal{H}, \varpi) \) may be viewed as commutative diagram in Cat:

\[ \begin{array}{ccc}
dom(a) & \xrightarrow{a} & S \\
F(h) & \downarrow \varpi & \\
dom(b) & \xrightarrow{b} & T. \\
\end{array} \]

Define functors

\[ \varpi_a : \mathcal{D}'(\mathcal{H}, \varpi) \to \text{Cat} \text{ via } (F(h), (a, b)) \mapsto \text{dom}(a) \text{ and} \]

\[ \varpi_b : \mathcal{D}'(\mathcal{H}, \varpi) \to \text{Cat} \text{ via } (F(h), (a, b)) \mapsto \text{dom}(b). \]
For these two functors select colimits (coproducts):

\[
\coprod \varpi_b(D) = \text{dom}(b) \quad \coprod \varpi_a(D) = \text{dom}(a)
\]

For each \( D = (\mathcal{F}(h), (a, b)) \) in \( \mathcal{D}'(\mathcal{H}, \varpi) \), \( \mathcal{F}(h) \in \textbf{Cat}(\text{dom}(a), \text{dom}(b)) \). By definition of colimit of \( \varpi_a \), these morphisms induce a unique \( \delta_\varpi \) as shown:

\[
\begin{array}{c}
\text{dom}(b) \xleftarrow{\mu_D} \coprod \varpi_b(D) \xrightarrow{\delta_\varpi} \coprod \varpi_a(D) \\
\text{dom}(a) \xleftarrow{\lambda_D}
\end{array}
\]

It can be shown that \( \delta_\varpi \in M' \) since each \( \mathcal{F}(h) \in M' \). The construction also gives the unique morphisms \( \alpha_\varpi \) and \( \beta_\varpi \) indicated in the diagrams below:

\[
\begin{array}{c}
\text{dom}(b) \xrightarrow{b} \coprod \varpi_b(D) \xrightarrow{\beta_\varpi} T \\
\text{dom}(a) \xrightarrow{a} \coprod \varpi_a(D)
\end{array}
\]

in \( \textbf{Cat} \)

Select a pushout in \( \textbf{Cat} \) of \( \alpha_\varpi \) and \( \delta_\varpi \):

\[
\begin{array}{c}
\coprod \varpi_a(D) \xrightarrow{\alpha_\varpi} S \\
\coprod \varpi_b(D) \xleftarrow{\beta_\varpi} \coprod \varpi_a(D) \xleftarrow{\delta_\varpi} S_{\varpi}
\end{array}
\]

Since \( \delta_\varpi \in M' \) and \( M' \) is stable under pushout \( u_\varpi \in M' \). However, for any \( D = (\mathcal{F}(h), (a, b)) \) in \( \mathcal{D}'(\mathcal{H}, \varpi) \),

\[
\varpi \circ \alpha_\varpi \circ \lambda_D = \varpi \circ a \\
= b \circ \mathcal{F}(h) \\
= \beta_\varpi \circ \mu_D \circ \mathcal{F}(h) \\
= \beta_\varpi \circ \delta_\varpi \circ \lambda_D.
\]

This implies \( \varpi \circ \alpha_\varpi = \beta_\varpi \circ \delta_\varpi \) and that there exists a unique \( p_\varpi \) for which the following is commutative in \( \textbf{Cat} \):

\[
\begin{array}{c}
\coprod \varpi_a(D) \xrightarrow{\alpha_\varpi} S \\
\coprod \varpi_b(D) \xleftarrow{\beta_\varpi} \coprod \varpi_a(D) \xleftarrow{\delta_\varpi} S_{\varpi}
\end{array}
\]
This gives the desired factorization $\varpi = p_{\varpi} \circ u_{\varpi}$ with $u_{\varpi} \in M'$.

11.248 Proof of CM5. Let $\mathcal{H}$ be a set of horns in $\hat{\Delta}$. For this choice each $\delta_{\varpi}$ and $u_{\varpi}$ obtained using the construction is in $W'$.

Let $f : A \to B$ be a functor between small categories. We may find a factorization of $f = p \circ i$ of $f$ with $p \in E'$ and $i \in W' \cap M'$ by iteration of the previous construction.

If $p_f$ were necessarily in $E'$, CM5 would be proved. However, the characterization 2.5 of $E'$ is not satisfied by $p_f$. Define

\[
\begin{align*}
A_0 &= A \\
A_1 &= Af, \\
A_2 &= A_{p_0}, \\
A_3 &= A_{p_1}, \\
& \vdots \\
p_{-1} &= f \\
p_0 &= p_f \\
p_1 &= p_{p_0} \\
p_2 &= p_{p_1} \\
& \vdots
\end{align*}
\]

These definitions are illustrated by:

This gives a functor $\mathcal{A} : (\mathbb{N}, \leq) \to \textbf{Cat}$ via $\mathcal{A}(n) = A_n$ and $\mathcal{A}(n, n) = 1_{A_n}$ for $n \in \mathbb{N}$, and $\mathcal{A}(m, n) = u_{n-1} \circ \cdots \circ u_m$ for $m \leq n$ in $\mathbb{N}$.

Select a colimit of this functor:

By 4.7.2(i) $i_0 \in M'$. By 4.7.2(ii) $i_0 \in W'$.

Define $p$ to be the unique morphism making all diagrams:

\[
\begin{align*}
A_n &\xrightarrow{i_n} A_{\infty} \\
B &\xleftarrow{p_n} A_{\infty}
\end{align*}
\]
in $\textbf{Cat}$ commutative. Then $f = p_{-1} = p \circ i_0$. This $p$ is in $E'$.

\vspace{1em}

11.249 Proofs of CM$_{52}$ and CM$_{42}$. Let $\mathcal{H}'$ be a set of canonical inclusions in $\hat{\Delta}$. Note that for this choice of $\mathcal{H}'$ the morphism $i$ constructed as on the previous page is in $M'$ and $p$ is in $W' \cap E'$. The factorization CM$_{51}$ and the construction are used to prove CM$_{42}$. □

11.3 Conclusions

As we have tried to show, one must look at Kan extensions as the tip of the iceberg of all the mathematical phenomena that is out there. We have three “dimensions” of such constructions:

- Left or Right
- Extension or Lifting
- Classical or Homotopy.

This gives us a total of eight different types of Kan phenomena to examine. For each of the eight we should look at different examples, classical algorithms that can implement the constructions, and, of course, quantum algorithms that can implement the constructions.

In Chapter 10 of Saunders Mac Lane’s classic textbook \cite{MacLane} \textit{Categories for a Working Mathematician} he titles one of his sections as

\textbf{All concepts are Kan extensions}

In that section he shows that adjoints, ends and coends, limits and colimits, etc. are all examples of Kan extensions. It is our strong belief that all these diverse types of Kan phenomena will be of use in many different areas of mathematics, computer science and physics. And it is in this ubiquity that we think there is a firm basis for its further study in quantum computing.
Appendix A

Axioms of set theory

There are over 6.2 billion humans inhabiting our planet. Each mind receives a persistent stream of information through various senses and strives to understand its data as some useful whole. A set is an aggregate, lump, pile, heap, collection, family, or number of things that belong or are used together. The abstract notions of set and number are closely related. The things which a set contains have various names: elements, members, individuals, . . .

The theory of sets is a first-order theory involving a single sort, sets and a single binary relation symbol $\varepsilon$ representing membership (see Chapter ??). The six axioms described in this Appendix characterize sets and the membership relation. They are: extent, powerset, union, infinity, foundation, and replacement. From these follow the redundant axioms: null set, comprehension, pairing as well as many constructions such as cartesian products. Extent and Regularity characterize the qualities of being a set. Power Set, Union, and Replacement give procedures for constructing new sets from given sets. The Axiom of Infinity asserts existence of a set.

A.1 Axiom of Extent

This first axiom of set theory is the formula

$$\vdash_{u,v} \forall x (x \in u \iff x \in v) \Rightarrow u = v$$

which asserts that a set is determined by its members. The formula above is short-hand for

$$\vdash_{u,v} \forall x (x \in u \Rightarrow x \in v) \land (x \in v \Rightarrow x \in u) \Rightarrow u = v$$

With these observations in mind, the Axiom of Extent reads: ‘If $u$ and $v$ are any sets, then $u = v$ if every $u$-element is a $v$-element and every $v$-element is a $u$-element.’ If $u$ is empty, then $w \in u$ is false for any $w$, so, $w \in v$ must be false. Similarly, if $v$ has no members then $u$ doesn’t either. In other words, if the hypothesis is a true statement and either $u$ or $v$ is the empty set (existence and uniqueness of which are proved in [121]), then both sets are empty. This axiom is not an operation for constructing new sets. We can not use it to conclude that any sets must exist. The aim of Extent is to characterize one property of being a set: the contents of the set are important but their order or arrangement is not. $\{0, 1\}$ and $\{1, 0\}$ differ typographically, for example, but represent the same set.
A.101 Our mathematical theory of sets has only one type of things, i.e. sets. This fact can be confusing to people who have worked with sets but have not studied an axiomatic development of their theory. In the expression \( \{0, 1\} \), for example, 0 and 1 must represent sets. The definitions \( 0 = \varnothing \) (the empty set) and \( 1 = \{\varnothing\} \) are convenient, have historical precedents, and can be justified using the axioms of Infinity, Replacement, and Extent. To see that Extent alone is inadequate, notice that this axiom does not permit us to conclude that \( \varnothing \) (or any other set) exists!

A.102 The converse of the Axiom of Extent is a theorem: \( u = v \Rightarrow \forall w (w \in u \iff w \in v) \).

Because: Assume \( u = v \). Let \( w \) be any set. Applying the substitution rule \((\land =)\) twice to \( w \in u \Rightarrow w \in u \) yields \( w \in u \Rightarrow w \in v \) and \( w \in v \Rightarrow w \in u \). Together these imply \( w \in u \iff w \in v \). Since \( w \) was arbitrary we may conclude \( \forall w (w \in u \iff w \in v) \).

A.103 A set \( u \) is a subset of a set \( v \) if every element of \( u \) is also a member of \( v \). The formula

\[
(u \subset v) \iff \forall x (x \in u \Rightarrow x \in v)
\]

defines the relation \( \subset \) which holds between certain pairs of sets. A consequence of the Axiom of Extent is the formula \( (u = v) \iff (u \subset v \land v \subset u) \).

A.104 \( (\top \vdash u \subset u) \).

Because: see [121].

A.2 Power Set Axiom

If \( u \) is any set, then there is also a set containing all the subsets of \( u \):

\[
(\top \vdash \exists x \forall y (\forall z (z \in y \Rightarrow z \in u) \Rightarrow y \in x))
\]

Note the use of the Convention for Presenting Theories. The formula asserts existence of a set, represented by the variable \( x \). Moreover, the axiom gives a condition for membership in \( x \): a thing is in \( x \) if that thing is a subset of \( u \). As it is written here, the axiom leaves open the possibility that \( x \) might contain other things as well. If, for example, \( u = \{0, 1\} \) and \( x = \{\varnothing, \{0\}, \{1\}, u, 2\} \), then \( \forall y (y \subset u \Rightarrow y \in x) \) is a true statement. The Axiom of Replacement will be used together with this Power Set Axiom to show that there is a set containing all the subsets of \( u \) and which does not contain extra stuff. The Axiom of Extent establishes that for any given set \( u \), this set of all subsets of \( u \) is unique.

A.3 Union Axiom

Membership, \( \epsilon \), is a relation between sets. If \( u \) is a set, its elements must therefore be sets. The Union Axiom asserts that given a set \( u \), there is a set which includes all those things which are contained in some member of \( u \):

\[
(\top \vdash \exists x \forall y (\forall z (z \in u \land y \in z) \Rightarrow y \in x))
\]

It provides another way of constructing a new set from a given set \( u \). To understand the formula, note that it asserts existence of a set represented by the variable \( x \). It also gives a sufficient condition for
A.4 Axiom of Infinity

A natural question to ask at this point is ‘Must there be any sets at all?’ The Axiom of Infinity asserts that yes, there must be a set, in fact, there must be an infinite one. Given this one set, other axioms permit us to construct a vast array of new sets. Here is a statement of the axiom:

$$(\top \vdash \exists x (\exists w (w \in x) \land \forall y (y \in x \Rightarrow \exists z (z \in x \land y \in z))))$$

It asserts existence of a set, denoted by the variable $x$. Notice that the formula in the scope of the outer-most quantifier is a conjunction of two simpler formulas. The first part asserts that $x$ must contain something: $\exists w (w \in x)$. The second part of the conjunction basically says that $x$ contains an endless supply of things. This is stated as follows: for any member $y$ of $x$, there is some other member $z \in x$ (which itself contains $y$). A possible structure for such a set $x$ can be depicted as follows. Suppose $x$ contains some element $y_0$. By the axiom, $y_0 \in y_1$ for some $y_1 \in x$. Continuation of this process yields an ascending chain:

$$y_0 \in y_1 \in \cdots \in x.$$ 

Must $y$ and its corresponding $z$ be distinct? We cannot conclude this without the Axiom of Foundation.

A.5 Axiom of Replacement

Let $\mathbb{N} = \{0, 1, 2, \ldots \}$ denote the set of natural numbers. The formula $\psi[m] \iff \exists n (m = 2n)$ expresses the property of ‘evenness.’ It seems reasonable that we should be able to extract from $\mathbb{N}$ the subset consisting of all the even numbers, that is, all the numbers $x$ for which the formula $\psi[y]$ is true. The Axiom of Replacement allows us to make such extractions from previously given sets. In fact, it is more general than that. Let $u$ be a set and let $\varphi[x, y]$ be a formula involving two variables.

$$((r \in u \land \varphi[r, s] \land \varphi[r, t] \Rightarrow s = t) \vdash_{r,s,t} \exists y (y \in x \iff \exists z (z \in u \land \varphi[z, y])))$$

This is a lengthy but important axiom so let us take some time to study the formula. The left side of the sequent expresses a property of the relation $\varphi$. It is useful to think of this property as the vertical line test that students meet in high school mathematics. Given any $r \in u$, there is at most one $y$ for which $\varphi[r, y]$ can be true.

A.501 The Comprehension Axiom can be obtained from Replacement. Let $\varphi[x, y]$ be the formula $(x = y) \land \psi[y]$. If $\varphi[x, y] \land \varphi[x, z]$ then $(x = y)$ and $(x = z)$ which implies $x = z$. So, the formula $\varphi[x, y]$ satisfies the hypothesis in the axiom of replacement. Consequently, for any formula $\psi[y]$ we may conclude

$$\exists x \forall y (y \in u \iff \exists z (z \in u \land \varphi[z, y]))$$

$$\exists x \forall y (y \in x \iff \exists z (z \in u \land (z = y) \land \psi[y]))$$

$$\exists x \forall y (y \in x \iff (y \in u \land \psi[y]))$$
A.502  We have used $\phi$ to represent the empty set but have not yet proved the existence or uniqueness of this set. To prove existence, let $u$ be the set whose existence is asserted by the Axiom of Infinity. Let $\psi[y]$ be the formula $y \neq y$. Then $\phi = \{y \in u : \psi[y]\}$. Notice that $\phi$ and $\{\phi\}$ are different sets. Also, $\mathcal{P}(\phi) = \{\phi\}$, $\mathcal{P}(\mathcal{P}(\phi)) = \{\phi, \{\phi\}\}$, etc. The Axiom of Extent establishes uniqueness so we may define the empty set to be the unique set having no members.

A.503  If $v$ and $w$ are sets, then there is a set, denoted $\{v, w\}$, having $v$ and $w$ as its only members.
Because: $u = \mathcal{P}^2 = \{\phi, \{\phi\}\}$ exists by the Power Set Axiom. The formula $\varphi[r, s]$ defined by

$$(r = \phi \land s = v ) \lor (r = \{\phi\} \land s = w)$$

satisfies the hypothesis in Replacement. Consequently, there is a set $x$ satisfying:

$$\forall y (y \in x \iff \exists z (z \in u \land \varphi[z, y])).$$

$\varphi[\phi, v]$ and $\varphi[\{\phi\}, w]$ are true, so, $v$ and $w$ are in $x$. If $y \neq v \land y \neq w$ then $\varphi[\phi, y]$ and $\varphi[\{\phi\}, y]$ are false, hence, $y \not\in x$.

A.504  By the Axiom of Extent, if $u$ is a set, there can be at most one set having the subsets of $u$ as its members. By the Power Set Axiom, there is at least one such set. Consequently, given any set $u$ there is exactly one set, denoted $\mathcal{P}(u)$ and called the power set of $u$, having the subsets of $u$ as its elements.

A.505  $(\top \vdash x, u \in \mathcal{P}(u) \iff x \subset u)$
Because: see [121].

A.506  $(\top \vdash u \in \mathcal{P}(u))$
Because: $x \in u \Rightarrow x \in u$. The definition of $\subset$ then implies $u \subset u$.

A.507  $(\top \vdash x, u \Rightarrow x \subset \bigcup u)$
Because: see [121].

A.508  $(\top \vdash y, u \in \bigcup u \iff \exists x (x \in u \land y \in x))$
Because: see [121].

A.509  Let $u$ and $v$ be sets. The union of $u$ and $v$ is the set containing those things which are in $u$, in $v$, or in both sets. It is denoted $u \cup v$ and is defined by

$$u \cup v = \bigcup \{u, v\}.$$
Let \( u \) and \( v \) be sets. The intersection of \( u \) and \( v \) is denoted \( u \cap v \) and is defined by
\[
 u \cap v = \{ x \in u : x \in v \}.
\]
Note that this is equal to \( \{ x \in v : x \in u \} \).

Let \( x \in X \) and \( y \in Y \). Then \( \{ x \} \), \( \{ x, y \} \in \mathcal{P}(X \cup Y) \). The ordered pair of \( x \) and \( y \) is denoted \( (x, y) \) and defined by
\[
(x, y) = \{ \{ x \}, \{ x, y \} \} \in \mathcal{P}(\mathcal{P}(X \cup Y)).
\]
The cartesian product of \( X \) and \( Y \) is the set of all such ordered pairs:
\[
X \times Y = \{ z \in \mathcal{P}^2(X \cup Y) : z = (x, y) \text{ for some } x \in X \text{ and } y \in Y \}.
\]

**A.6 Axiom of Foundation**

Like the Axiom of Extent, the sequent
\[
(u \neq \phi \vdash_u \exists y (y \in u \land (y \cap u = \phi)))
\]
characterizes a quality of sets. To understand this axiom, recall that set theory involves a single type of thing (i.e. sets) and the membership relation which holds between some pairs of sets. Let \( u_0 \) be a set. If \( u_0 \) isn’t empty, then there is a set \( u_1 \) for which \( u_1 \in u_0 \). Similarly, \( u_2 \in u_1 \in u_0 \) unless \( u_1 = \phi \). Unless we eventually run into the empty set, there must be a descending chain \( \cdots \in u_n \in u_{n-1} \in \cdots \in u_1 \in u_0 \).
The Axiom of Foundation asserts that any such chain must stop. Essentially, it says that all sets are built up from \( \phi \) in some way.

**A.601 No set is a member of itself: \( x \not\in x \).**

*Because:* if \( u = \{ x \} \) then \( y \in u \) implies \( y = x \). Consequently, \( y \cap u = x \cap \{ x \} \). If \( x \in x \) then \( x \in x \cap \{ x \} \), contradicting the fact that some \( y \) in \( u \) must have empty intersection with \( u \).

**A.602 For no sets \( x \) and \( z \) is \( x \in z \in x \) true.**

*Because:* if \( u = \{ x, z \} \) then \( y \in u \) implies \( y = x \) or \( y = z \). In the former case, \( y \cap u = x \cap \{ x, z \} \) which would contain \( z \) if \( z \in x \). In the latter case, \( y \cap u = z \cap \{ x, z \} \) which would contain \( x \) if \( x \in z \).
Consequently, if \( x \in z \in x \) then \( y \cap u \) must be nonempty, contradicting the Axiom of Foundation.
Appendix B

Benchmark Input and Output from Left Kan Extension Implementation

XML File Used to Specify the Domain Category: a Discrete Category with Three Objects

```xml
<?xml version="1.0" encoding="UTF-8"?>
<category xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNameSpaceSchemaLocation="category-schema.xsd">
    <name>Discrete category from Example 9 on page 156 of Bob Walters' book</name>
    <objects>3</objects>
    <generatorList>
    </generatorList>
    <relationList>
    </relationList>
</category>
```
XML File Used to Specify the Codomain Category

<?xml version="1.0" encoding="UTF-8"?>
<category xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNameSpaceSchemaLocation="category-schema.xsd">
  <name>Example 9 on page 156 of Bob Walters’ book</name>
  <objects>3</objects>
  <generatorList>
    <generator>
      <name>a</name>
      <domain>0</domain>
      <codomain>1</codomain>
    </generator>
    <generator>
      <name>b</name>
      <domain>1</domain>
      <codomain>2</codomain>
    </generator>
    <generator>
      <name>c</name>
      <domain>2</domain>
      <codomain>0</codomain>
    </generator>
  </generatorList>
  <relationList>
    <relation>
      <path>
        <leg>a</leg>
        <leg>b</leg>
        <leg>c</leg>
      </path>
    </relation>
    <relation>
      <path>
        <leg>b</leg>
        <leg>c</leg>
        <leg>a</leg>
      </path>
    </relation>
    <relation>
      <path>
        <leg>c</leg>
        <leg>a</leg>
        <leg>b</leg>
      </path>
    </relation>
  </relationList>
</category>
XML File Used to Specify the Functor: $A \to B$

```xml
<?xml version="1.0" encoding="UTF-8"?>

<functor xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNameSpaceSchemaLocation="category-schema.xsd">
  <name>Functor from example 9 on page 156 of Walters' book</name>
  <objectImages>
    <objectImage>
      <input>0</input>
      <output>0</output>
    </objectImage>
    <objectImage>
      <input>1</input>
      <output>1</output>
    </objectImage>
    <objectImage>
      <input>2</input>
      <output>2</output>
    </objectImage>
  </objectImages>
  <generatorImages>
    <generatorImages>
    </generatorImages>
  </generatorImages>
</functor>
```
XML File Used to Specify the Action: \( A \rightarrow \text{Set} \)

```xml
<?xml version="1.0" encoding="UTF-8"?>
<action xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNameSpaceSchemaLocation="action-schema.xsd">
  <name>Action from page 156 of Bob Walters' book.</name>
  <objectImages>
    <objectImage>
      <input>0</input>
      <output>1</output>
    </objectImage>
    <objectImage>
      <input>1</input>
      <output>1</output>
    </objectImage>
    <objectImage>
      <input>2</input>
      <output>1</output>
    </objectImage>
  </objectImages>
  <generatorImages>
  </generatorImages>
</action>
```
Benchmark Output from the Left Kan Extension Implementation

```
java -classpath cls org.bmrc.qkan.test.classical.AbacusTest;
*****************************************************************************
Computing left Kan extension with the following data:
Domain category.
Name     : Discrete category from Example 9 on page 156 of Bob Walters’ book
Objects  : 3
Generators:
Relations :
Codomain category.
Name     : Example 9 on page 156 of Bob Walters’ book
Objects  : 3
Generators:
   a:0->1
   b:1->2
   c:2->0
Relations :
   (a:0 -> 1)(b:1 -> 2)(c:2 -> 0) = id
   (b:1 -> 2)(c:2 -> 0)(a:0 -> 1) = id
   (c:2 -> 0)(a:0 -> 1)(b:1 -> 2) = id
Functor.
Functor  : Functor from example 9 on page 156 of Walters’ book
   Domain     : Discrete category from Example 9 on page 156 of Bob Walters’ book
   Codomain   : Example 9 on page 156 of Bob Walters’ book
   Object images :
      0: 0
      1: 1
      2: 2
   Generator images:
Action.
Action from page 156 of Bob Walters’ book.
Domain
Name     : Discrete category from Example 9 on page 156 of Bob Walters’ book
Objects  : 3
Generators:
Relations :
Action definition
Objects
   0 : 1
   1 : 1
   2 : 1
Morphisms
*****************************************************************************
Entering initializeTables() method.
Epsilon Tables:
   epsilon_0 [ -1 ]
```
epsilon_1 [ -1 ]
epsilon_2 [ -1 ]
L Tables:
L table for 0 initialized to
L(0)  a
L table for 1 initialized to
L(1)  b
L table for 2 initialized to
L(2)  c
Relation Tables:
Table for relation: (c:2 -> 0)(a:0 -> 1)(b:1 -> 2) = id
Left side of the table:
L(2)  c  a  b
Right side of the table:
L(2)
Table for relation: (a:0 -> 1)(b:1 -> 2)(c:2 -> 0) = id
Left side of the table:
L(0)  a  b  c
Right side of the table:
L(0)
Table for relation: (b:1 -> 2)(c:2 -> 0)(a:0 -> 1) = id
Left side of the table:
L(1)  b  c  a
Right side of the table:
L(1)
Naturality Tables:
Exiting initializeTables() method.
Entering undefinedElements() method.
Exiting undefinedElements() method.
Entering defineNewElement() method.
Added element 0 to L(0) in epsilon table 0
Epsilon Table for 0 is now
epsilon_0 [ 0 ]
L Table for 0 is now
L(0)  a
  0  -1
Exiting defineNewElement() method.
Entering fillInConsequences() method.
Working with relation: (c:2 -> 0)(a:0 -> 1)(b:1 -> 2) = id
Here is the relation table (c:2 -> 0)(a:0 -> 1)(b:1 -> 2) = id after filling in consequences.
Table for relation: (c:2 -> 0)(a:0 -> 1)(b:1 -> 2) = id
Left side of the table:
L(2)  c  a  b
Right side of the table:
L(2)
Working with relation: (a:0 -> 1)(b:1 -> 2)(c:2 -> 0) = id
Here is the relation table (a:0 -> 1)(b:1 -> 2)(c:2 -> 0) = id after filling in consequences.
Table for relation: \( (a:0 \rightarrow 1)(b:1 \rightarrow 2)(c:2 \rightarrow 0) = id \)

Left side of the table:

<table>
<thead>
<tr>
<th>L(0)</th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>

Right side of the table:

<table>
<thead>
<tr>
<th>L(0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

Working with relation: \( (b:1 \rightarrow 2)(c:2 \rightarrow 0)(a:0 \rightarrow 1) = id \)

Here is the relation table \( (b:1 \rightarrow 2)(c:2 \rightarrow 0)(a:0 \rightarrow 1) = id \) after filling in consequences.

Table for relation: \( (b:1 \rightarrow 2)(c:2 \rightarrow 0)(a:0 \rightarrow 1) = id \)

Left side of the table:

<table>
<thead>
<tr>
<th>L(1)</th>
<th>b</th>
<th>c</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Right side of the table:

<table>
<thead>
<tr>
<th>L(1)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

Exiting fillInConsequences() method.

Entering coincidencesExist() method.

Exiting coincidencesExist() method.

Entering undefinedElements() method.

Exiting undefinedElements() method.

Entering defineNewElement() method.

Added element 0 to L(1) in epsilon table 1

Epsilon Table for 1 is now

epsilon_1 [ 0 ]

L Table for 1 is now

<table>
<thead>
<tr>
<th>L(1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>b</td>
</tr>
</tbody>
</table>

Exiting defineNewElement() method.

Entering fillInConsequences() method.

Working with relation: \( (c:2 \rightarrow 0)(a:0 \rightarrow 1)(b:1 \rightarrow 2) = id \)

Here is the relation table \( (c:2 \rightarrow 0)(a:0 \rightarrow 1)(b:1 \rightarrow 2) = id \) after filling in consequences.

Table for relation: \( (c:2 \rightarrow 0)(a:0 \rightarrow 1)(b:1 \rightarrow 2) = id \)

Left side of the table:

<table>
<thead>
<tr>
<th>L(2)</th>
<th>c</th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Right side of the table:

<table>
<thead>
<tr>
<th>L(2)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

Working with relation: \( (a:0 \rightarrow 1)(b:1 \rightarrow 2)(c:2 \rightarrow 0) = id \)

Here is the relation table \( (a:0 \rightarrow 1)(b:1 \rightarrow 2)(c:2 \rightarrow 0) = id \) after filling in consequences.

Table for relation: \( (a:0 \rightarrow 1)(b:1 \rightarrow 2)(c:2 \rightarrow 0) = id \)

Left side of the table:

<table>
<thead>
<tr>
<th>L(0)</th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>

Right side of the table:

<table>
<thead>
<tr>
<th>L(0)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

Working with relation: \( (b:1 \rightarrow 2)(c:2 \rightarrow 0)(a:0 \rightarrow 1) = id \)

Here is the relation table \( (b:1 \rightarrow 2)(c:2 \rightarrow 0)(a:0 \rightarrow 1) = id \) after filling in consequences.

Table for relation: \( (b:1 \rightarrow 2)(c:2 \rightarrow 0)(a:0 \rightarrow 1) = id \)
Left side of the table:
L(1)  b  c  a
   0  -1  -1  -1
Right side of the table:
L(1)
   0

Exiting fillInConsequences() method.
Entering coincidencesExist() method.
Exiting coincidencesExist() method.
Entering undefinedElements() method.
Exiting undefinedElements() method.
Entering defineNewElement() method.
Added element 0 to L(2) in epsilon table 2
Epsilon Table for 2 is now
epsilon_2 [ 0 ]
L Table for 2 is now
L(2)  c
   0  -1

Exiting defineNewElement() method.
Entering fillInConsequences() method.
Working with relation: (c:2 -> 0)(a:0 -> 1)(b:1 -> 2) = id
Here is the relation table (c:2 -> 0)(a:0 -> 1)(b:1 -> 2) = id after filling in consequences.
Table for relation: (c:2 -> 0)(a:0 -> 1)(b:1 -> 2) = id
Left side of the table:
L(2)  c  a  b
   0  -1  -1  -1
Right side of the table:
L(2)
   0

Working with relation: (a:0 -> 1)(b:1 -> 2)(c:2 -> 0) = id
Here is the relation table (a:0 -> 1)(b:1 -> 2)(c:2 -> 0) = id after filling in consequences.
Table for relation: (a:0 -> 1)(b:1 -> 2)(c:2 -> 0) = id
Left side of the table:
L(0)  a  b  c
   0  -1  -1  -1
Right side of the table:
L(0)
   0

Working with relation: (b:1 -> 2)(c:2 -> 0)(a:0 -> 1) = id
Here is the relation table (b:1 -> 2)(c:2 -> 0)(a:0 -> 1) = id after filling in consequences.
Table for relation: (b:1 -> 2)(c:2 -> 0)(a:0 -> 1) = id
Left side of the table:
L(1)  b  c  a
   0  -1  -1  -1
Right side of the table:
L(1)
   0
Exiting `fillInConsequences()` method.
Entering `coincidencesExist()` method.
Exiting `coincidencesExist()` method.
Entering `undefinedElements()` method.
Exiting `undefinedElements()` method.
Entering `defineNewElement()` method.
Added element 1 to L(1) in L table for 0
L Table for 0 is now
L(0) | a
| 0 1
L Table for 1 is now
L(1) | b
| 0 -1
| 1 -1
Exiting `defineNewElement()` method.
Entering `fillInConsequences()` method.
Working with relation: \( (c:2 \rightarrow 0)(a:0 \rightarrow 1)(b:1 \rightarrow 2) = \text{id} \)
Here is the relation table \( (c:2 \rightarrow 0)(a:0 \rightarrow 1)(b:1 \rightarrow 2) = \text{id} \) after filling in consequences.
Table for relation: \( (c:2 \rightarrow 0)(a:0 \rightarrow 1)(b:1 \rightarrow 2) = \text{id} \)
Left side of the table:
L(2) | c | a | b
| 0 | -1 | -1 | -1
Right side of the table:
L(2) | 0
Working with relation: \( (a:0 \rightarrow 1)(b:1 \rightarrow 2)(c:2 \rightarrow 0) = \text{id} \)
Here is the relation table \( (a:0 \rightarrow 1)(b:1 \rightarrow 2)(c:2 \rightarrow 0) = \text{id} \) after filling in consequences.
Table for relation: \( (a:0 \rightarrow 1)(b:1 \rightarrow 2)(c:2 \rightarrow 0) = \text{id} \)
Left side of the table:
L(0) | a | b | c
| 0 | 1 | -1 | -1
Right side of the table:
L(0) | 0
Working with relation: \( (b:1 \rightarrow 2)(c:2 \rightarrow 0)(a:0 \rightarrow 1) = \text{id} \)
Here is the relation table \( (b:1 \rightarrow 2)(c:2 \rightarrow 0)(a:0 \rightarrow 1) = \text{id} \) after filling in consequences.
Table for relation: \( (b:1 \rightarrow 2)(c:2 \rightarrow 0)(a:0 \rightarrow 1) = \text{id} \)
Left side of the table:
L(1) | b | c | a
| 0 | -1 | -1 | -1
| 1 | -1 | -1 | -1
Right side of the table:
L(1) | 0
| 1
Exiting `fillInConsequences()` method.
Entering `coincidencesExist()` method.
Exiting coincidencesExist() method.
Entering undefinedElements() method.
Exiting undefinedElements() method.
Entering defineNewElement() method.
Added element 1 to L(2) in L table for 1
L Table for 1 is now
L(1) b
  0  1
  1 -1

L Table for 2 is now
L(2) c
  0 -1
  1 -1

Exiting defineNewElement() method.
Entering fillInConsequences() method.
Working with relation: (c:2 -> 0)(a:0 -> 1)(b:1 -> 2) = id
Here is the relation table (c:2 -> 0)(a:0 -> 1)(b:1 -> 2) = id after filling in consequences.
Table for relation: (c:2 -> 0)(a:0 -> 1)(b:1 -> 2) = id
Left side of the table:
L(2) c a b
  0 -1 -1 -1
  1 -1 -1 -1
Right side of the table:
L(2)
  0
  1

Working with relation: (a:0 -> 1)(b:1 -> 2)(c:2 -> 0) = id
Here is the relation table (a:0 -> 1)(b:1 -> 2)(c:2 -> 0) = id after filling in consequences.
Table for relation: (a:0 -> 1)(b:1 -> 2)(c:2 -> 0) = id
Left side of the table:
L(0) a b c
  0  1 -1 -1
Right side of the table:
L(0)
  0

Working with relation: (b:1 -> 2)(c:2 -> 0)(a:0 -> 1) = id
Here is the relation table (b:1 -> 2)(c:2 -> 0)(a:0 -> 1) = id after filling in consequences.
Table for relation: (b:1 -> 2)(c:2 -> 0)(a:0 -> 1) = id
Left side of the table:
L(1) b c a
  0  1 -1 -1
  1 -1 -1 -1
Right side of the table:
L(1)
  0
  1

Exiting fillInConsequences() method.
Entering coincidencesExist() method.
Exiting coincidencesExist() method.
Entering undefinedElements() method.
Exiting undefinedElements() method.
Entering defineNewElement() method.

Added element 2 to L(2) in L table for 1

<table>
<thead>
<tr>
<th>L Table for 1 is now</th>
</tr>
</thead>
<tbody>
<tr>
<td>L(1)</td>
</tr>
<tr>
<td>b</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
</tbody>
</table>

L Table for 2 is now

<table>
<thead>
<tr>
<th>L(2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>-1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>-1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>-1</td>
</tr>
</tbody>
</table>

Exiting defineNewElement() method.
Entering fillInConsequences() method.

Working with relation: (c:2 → 0)(a:0 → 1)(b:1 → 2) = id

Here is the relation table (c:2 → 0)(a:0 → 1)(b:1 → 2) = id after filling in consequences.

<table>
<thead>
<tr>
<th>Table for relation: (c:2 → 0)(a:0 → 1)(b:1 → 2) = id</th>
</tr>
</thead>
<tbody>
<tr>
<td>Left side of the table:</td>
</tr>
<tr>
<td>L(2) c a b</td>
</tr>
<tr>
<td>0 -1 -1 -1</td>
</tr>
<tr>
<td>1 -1 -1 -1</td>
</tr>
<tr>
<td>2 -1 -1 -1</td>
</tr>
<tr>
<td>Right side of the table:</td>
</tr>
<tr>
<td>L(2)</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
</tbody>
</table>

Working with relation: (a:0 → 1)(b:1 → 2)(c:2 → 0) = id

Here is the relation table (a:0 → 1)(b:1 → 2)(c:2 → 0) = id after filling in consequences.

<table>
<thead>
<tr>
<th>Table for relation: (a:0 → 1)(b:1 → 2)(c:2 → 0) = id</th>
</tr>
</thead>
<tbody>
<tr>
<td>Left side of the table:</td>
</tr>
<tr>
<td>L(0) a b c</td>
</tr>
<tr>
<td>0 1 2 -1</td>
</tr>
<tr>
<td>Right side of the table:</td>
</tr>
<tr>
<td>L(0)</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

Working with relation: (b:1 → 2)(c:2 → 0)(a:0 → 1) = id

Here is the relation table (b:1 → 2)(c:2 → 0)(a:0 → 1) = id after filling in consequences.

<table>
<thead>
<tr>
<th>Table for relation: (b:1 → 2)(c:2 → 0)(a:0 → 1) = id</th>
</tr>
</thead>
<tbody>
<tr>
<td>Left side of the table:</td>
</tr>
<tr>
<td>L(1) b c a</td>
</tr>
<tr>
<td>0 1 -1 -1</td>
</tr>
<tr>
<td>1 2 -1 -1</td>
</tr>
<tr>
<td>Right side of the table:</td>
</tr>
<tr>
<td>L(1)</td>
</tr>
</tbody>
</table>
Exiting fillInConsequences() method.
Entering coincidencesExist() method.
Exiting coincidencesExist() method.
Entering undefinedElements() method.
Exiting undefinedElements() method.
Entering defineNewElement() method.
Added element 1 to L(0) in L table for 2

L Table for 2 is now

<table>
<thead>
<tr>
<th>L(2)</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>-1</td>
</tr>
<tr>
<td>2</td>
<td>-1</td>
</tr>
</tbody>
</table>

L Table for 0 is now

<table>
<thead>
<tr>
<th>L(0)</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>-1</td>
</tr>
</tbody>
</table>

Exiting defineNewElement() method.
Entering fillInConsequences() method.

Working with relation: (c:2 -> 0)(a:0 -> 1)(b:1 -> 2) = id

Here is the relation table (c:2 -> 0)(a:0 -> 1)(b:1 -> 2) = id after filling in consequences.

Table for relation: (c:2 -> 0)(a:0 -> 1)(b:1 -> 2) = id
Left side of the table:

<table>
<thead>
<tr>
<th>L(2)</th>
<th>c</th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>2</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>

Right side of the table:

<table>
<thead>
<tr>
<th>L(2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
</tbody>
</table>

Working with relation: (a:0 -> 1)(b:1 -> 2)(c:2 -> 0) = id

Here is the relation table (a:0 -> 1)(b:1 -> 2)(c:2 -> 0) = id after filling in consequences.

Table for relation: (a:0 -> 1)(b:1 -> 2)(c:2 -> 0) = id
Left side of the table:

<table>
<thead>
<tr>
<th>L(0)</th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>-1</td>
</tr>
<tr>
<td>1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>

Right side of the table:

<table>
<thead>
<tr>
<th>L(0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

Working with relation: (b:1 -> 2)(c:2 -> 0)(a:0 -> 1) = id

Here is the relation table (b:1 -> 2)(c:2 -> 0)(a:0 -> 1) = id after filling in consequences.

Table for relation: (b:1 -> 2)(c:2 -> 0)(a:0 -> 1) = id
Left side of the table:
L(1) b c a
  0 1 -1 -1
  1 2 -1 -1
Right side of the table:
L(1)
  0
  1

Exiting fillInConsequences() method.
Entering coincidencesExist() method.
Exiting coincidencesExist() method.
Entering undefinedElements() method.
Exiting undefinedElements() method.
Entering defineNewElement() method.
Added element 2 to L(1) in L table for 0
L Table for 0 is now
L(0) a
  0 1
  1 2
L Table for 1 is now
L(1) b
  0 1
  1 2
  2 -1

Exiting defineNewElement() method.
Entering fillInConsequences() method.
Working with relation: (c:2 -> 0)(a:0 -> 1)(b:1 -> 2) = id
Here is the relation table (c:2 -> 0)(a:0 -> 1)(b:1 -> 2) = id after filling in consequences.
Table for relation: (c:2 -> 0)(a:0 -> 1)(b:1 -> 2) = id
Left side of the table:
L(2) c a b
  0 1 2 -1
  1 -1 -1 -1
  2 -1 -1 -1
Right side of the table:
L(2)
  0
  1
  2

Working with relation: (a:0 -> 1)(b:1 -> 2)(c:2 -> 0) = id
Here is the relation table (a:0 -> 1)(b:1 -> 2)(c:2 -> 0) = id after filling in consequences.
Table for relation: (a:0 -> 1)(b:1 -> 2)(c:2 -> 0) = id
Left side of the table:
L(0) a b c
  0 1 2 -1
  1 2 -1 -1
Right side of the table:
L(0)
Working with relation: (b:1 -> 2)(c:2 -> 0)(a:0 -> 1) = id
Here is the relation table (b:1 -> 2)(c:2 -> 0)(a:0 -> 1) = id after filling in consequences.

<table>
<thead>
<tr>
<th>Left side of the table:</th>
</tr>
</thead>
<tbody>
<tr>
<td>L(1) b c a</td>
</tr>
<tr>
<td>0  1 -1 -1</td>
</tr>
<tr>
<td>1  2 -1 -1</td>
</tr>
<tr>
<td>2 -1 -1 -1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Right side of the table:</th>
</tr>
</thead>
<tbody>
<tr>
<td>L(1)</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
</tbody>
</table>

Exiting fillInConsequences() method.
Entering coincidencesExist() method.
Exiting coincidencesExist() method.
Entering undefinedElements() method.
Exiting undefinedElements() method.
Entering defineNewElement() method.
Added element 3 to L(2) in L table for 1
L Table for 1 is now

<table>
<thead>
<tr>
<th>L(1) b</th>
</tr>
</thead>
<tbody>
<tr>
<td>0  1</td>
</tr>
<tr>
<td>1  2</td>
</tr>
<tr>
<td>2  3</td>
</tr>
</tbody>
</table>

L Table for 2 is now

<table>
<thead>
<tr>
<th>L(2) c</th>
</tr>
</thead>
<tbody>
<tr>
<td>0  1</td>
</tr>
<tr>
<td>1 -1</td>
</tr>
<tr>
<td>2 -1</td>
</tr>
<tr>
<td>3 -1</td>
</tr>
</tbody>
</table>

Exiting defineNewElement() method.
Entering fillInConsequences() method.
Working with relation: (c:2 -> 0)(a:0 -> 1)(b:1 -> 2) = id
Here is the relation table (c:2 -> 0)(a:0 -> 1)(b:1 -> 2) = id after filling in consequences.

<table>
<thead>
<tr>
<th>Left side of the table:</th>
</tr>
</thead>
<tbody>
<tr>
<td>L(2) c a b</td>
</tr>
<tr>
<td>0  1  2  3</td>
</tr>
<tr>
<td>1 -1 -1 -1</td>
</tr>
<tr>
<td>2 -1 -1 -1</td>
</tr>
<tr>
<td>3 -1 -1 -1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Right side of the table:</th>
</tr>
</thead>
<tbody>
<tr>
<td>L(2)</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>
Working with relation: (a:0 -> 1)(b:1 -> 2)(c:2 -> 0) = id
Here is the relation table (a:0 -> 1)(b:1 -> 2)(c:2 -> 0) = id after filling in consequences.
Table for relation: (a:0 -> 1)(b:1 -> 2)(c:2 -> 0) = id
Left side of the table:
L(0)  a  b  c
    0  1  2  -1
    1  2  3  -1
Right side of the table:
L(0)
    0
    1

Working with relation: (b:1 -> 2)(c:2 -> 0)(a:0 -> 1) = id
Here is the relation table (b:1 -> 2)(c:2 -> 0)(a:0 -> 1) = id after filling in consequences.
Table for relation: (b:1 -> 2)(c:2 -> 0)(a:0 -> 1) = id
Left side of the table:
L(1)  b  c  a
    0  1  -1  -1
    1  2  -1  -1
    2  3  -1  -1
Right side of the table:
L(1)
    0
    1
    2

Exiting fillInConsequences() method.
Entering coincidencesExist() method.
Added coincidence (2, 0, 3) from relation table for relation (c:2 -> 0)(a:0 -> 1)(b:1 -> 2) = id
Exiting coincidencesExist() method.
Entering dealWithCoincidences() method.
Exiting dealWithCoincidences() method.
Entering fillInConsequences() method.
Working with relation: (c:2 -> 0)(a:0 -> 1)(b:1 -> 2) = id
Here is the relation table (c:2 -> 0)(a:0 -> 1)(b:1 -> 2) = id after filling in consequences.
Table for relation: (c:2 -> 0)(a:0 -> 1)(b:1 -> 2) = id
Left side of the table:
L(2)  c  a  b
    0  1  2  0
    1 -1 -1 -1
    2 -1 -1 -1
    0  1  2  0
Right side of the table:
L(2)
    0
    1
    2
    0
Working with relation: \((a:0 \rightarrow 1)(b:1 \rightarrow 2)(c:2 \rightarrow 0) = \text{id}\)
Here is the relation table \((a:0 \rightarrow 1)(b:1 \rightarrow 2)(c:2 \rightarrow 0) = \text{id}\) after filling in consequences.
Table for relation: \((a:0 \rightarrow 1)(b:1 \rightarrow 2)(c:2 \rightarrow 0) = \text{id}\)
Left side of the table:
\[
\begin{array}{ccc}
L(0) & a & b & c \\
0 & 1 & 2 & -1 \\
1 & 2 & 0 & 1 \\
\end{array}
\]
Right side of the table:
\[
L(0)
\]
\[
0 \\
1
\]

Working with relation: \((b:1 \rightarrow 2)(c:2 \rightarrow 0)(a:0 \rightarrow 1) = \text{id}\)
Here is the relation table \((b:1 \rightarrow 2)(c:2 \rightarrow 0)(a:0 \rightarrow 1) = \text{id}\) after filling in consequences.
Table for relation: \((b:1 \rightarrow 2)(c:2 \rightarrow 0)(a:0 \rightarrow 1) = \text{id}\)
Left side of the table:
\[
\begin{array}{ccc}
L(1) & b & c & a \\
0 & 1 & -1 & -1 \\
1 & 2 & -1 & -1 \\
2 & 0 & 1 & 2 \\
\end{array}
\]
Right side of the table:
\[
L(1)
\]
\[
0 \\
1 \\
2
\]

Exiting `fillInConsequences()` method.

Entering `coincidencesExist()` method.

Exiting `coincidencesExist()` method.

Entering `dealWithCoincidences()` method.

Exiting `dealWithCoincidences()` method.

Entering `fillInConsequences()` method.

Working with relation: \((c:2 \rightarrow 0)(a:0 \rightarrow 1)(b:1 \rightarrow 2) = \text{id}\)
Here is the relation table \((c:2 \rightarrow 0)(a:0 \rightarrow 1)(b:1 \rightarrow 2) = \text{id}\) after filling in consequences.
Table for relation: \((c:2 \rightarrow 0)(a:0 \rightarrow 1)(b:1 \rightarrow 2) = \text{id}\)
Left side of the table:
\[
\begin{array}{ccc}
L(2) & c & a & b \\
0 & 1 & 2 & 0 \\
1 & -1 & -1 & -1 \\
2 & -1 & -1 & -1 \\
0 & 1 & 2 & 0 \\
\end{array}
\]
Right side of the table:
\[
L(2)
\]
\[
0 \\
1 \\
2
\]

Working with relation: \((a:0 \rightarrow 1)(b:1 \rightarrow 2)(c:2 \rightarrow 0) = \text{id}\)
Here is the relation table \((a:0 \rightarrow 1)(b:1 \rightarrow 2)(c:2 \rightarrow 0) = \text{id}\) after filling in consequences.
Table for relation: \((a:0 \rightarrow 1)(b:1 \rightarrow 2)(c:2 \rightarrow 0) = \text{id}\)
Left side of the table:
Working with relation: (b:1 -> 2)(c:2 -> 0)(a:0 -> 1) = id
Here is the relation table (b:1 -> 2)(c:2 -> 0)(a:0 -> 1) = id after filling in consequences.

Table for relation: (b:1 -> 2)(c:2 -> 0)(a:0 -> 1) = id
Left side of the table:
L(1)  
<table>
<thead>
<tr>
<th>b</th>
<th>c</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>-1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>-1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Right side of the table:
L(1)
0
1
2

Exiting fillInConsequences() method.
Entering coincidencesExist() method.
Exiting coincidencesExist() method.
Entering undefinedElements() method.
Exiting undefinedElements() method.
Entering defineNewElement() method.
Added element 2 to L(0) in L table for 2
L Table for 2 is now
L(2)  
<table>
<thead>
<tr>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>
L Table for 0 is now
L(0)  
<table>
<thead>
<tr>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>2</td>
</tr>
</tbody>
</table>

Exiting defineNewElement() method.
Entering fillInConsequences() method.
Working with relation: (c:2 -> 0)(a:0 -> 1)(b:1 -> 2) = id
Here is the relation table (c:2 -> 0)(a:0 -> 1)(b:1 -> 2) = id after filling in consequences.
Table for relation: (c:2 -> 0)(a:0 -> 1)(b:1 -> 2) = id
Left side of the table:
L(2)  
<table>
<thead>
<tr>
<th>c</th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>-1</td>
</tr>
<tr>
<td>2</td>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>

Right side of the table:
L(2)
0
1
2
0

Working with relation: (a:0 -> 1)(b:1 -> 2)(c:2 -> 0) = id
Here is the relation table (a:0 -> 1)(b:1 -> 2)(c:2 -> 0) = id after filling in consequences.
Table for relation: (a:0 -> 1)(b:1 -> 2)(c:2 -> 0) = id
Left side of the table:
L(0) a b c
0 1 2 -1
1 2 0 1
2 -1 -1 -1
Right side of the table:
L(0)
0
1
2

Working with relation: (b:1 -> 2)(c:2 -> 0)(a:0 -> 1) = id
Here is the relation table (b:1 -> 2)(c:2 -> 0)(a:0 -> 1) = id after filling in consequences.
Table for relation: (b:1 -> 2)(c:2 -> 0)(a:0 -> 1) = id
Left side of the table:
L(1) b c a
0 1 2 -1
1 2 -1 -1
2 0 1 2
Right side of the table:
L(1)
0
1
2

Exiting fillInConsequences() method.
Entering coincidencesExist() method.
Exiting coincidencesExist() method.
Entering undefinedElements() method.
Exiting undefinedElements() method.
Entering defineNewElement() method.
Added element 3 to L(1) in L table for 0
L Table for 0 is now
L(0) a
0 1
1 2
2 3

L Table for 1 is now
L(1) b
0 1
Exiting `defineNewElement()` method.
Entering `fillInConsequences()` method.
Working with relation: (c:2 -> 0)(a:0 -> 1)(b:1 -> 2) = id
Here is the relation table (c:2 -> 0)(a:0 -> 1)(b:1 -> 2) = id after filling in consequences.
Table for relation: (c:2 -> 0)(a:0 -> 1)(b:1 -> 2) = id
Left side of the table:
L(2)   c  a  b
0     1  2  0
1     2  3 -1
2    -1 -1 -1
0     1  2  0
Right side of the table:
L(2)
  0
  1
  2
  0

Working with relation: (a:0 -> 1)(b:1 -> 2)(c:2 -> 0) = id
Here is the relation table (a:0 -> 1)(b:1 -> 2)(c:2 -> 0) = id after filling in consequences.
Table for relation: (a:0 -> 1)(b:1 -> 2)(c:2 -> 0) = id
Left side of the table:
L(0)   a  b  c
0     1  2 -1
1     2  0  1
2    -1 -1 -1
Right side of the table:
L(0)
  0
  1
  2

Working with relation: (b:1 -> 2)(c:2 -> 0)(a:0 -> 1) = id
Here is the relation table (b:1 -> 2)(c:2 -> 0)(a:0 -> 1) = id after filling in consequences.
Table for relation: (b:1 -> 2)(c:2 -> 0)(a:0 -> 1) = id
Left side of the table:
L(1)   b  c  a
0     1  2  3
1     2 -1 -1
2     0  1  2
3    -1 -1 -1
Right side of the table:
L(1)
  0
  1
  2
  3
Exiting `fillInConsequences()` method.
Entering `coincidencesExist()` method.
Added coincidence (1, 0, 3) from relation table for relation (b:1 -> 2)(c:2 -> 0)(a:0 -> 1) = id
Exiting `coincidencesExist()` method.
Entering `dealWithCoincidenes()` method.
Exiting `dealWithCoincidenes()` method.
Entering `fillInConsequences()` method.
Working with relation: (c:2 -> 0)(a:0 -> 1)(b:1 -> 2) = id
Here is the relation table (c:2 -> 0)(a:0 -> 1)(b:1 -> 2) = id after filling in consequences.
Table for relation: (c:2 -> 0)(a:0 -> 1)(b:1 -> 2) = id
Left side of the table:
L(2)  c  a  b
   0  1  2  0
   1  2  0  1
   2 -1 -1 -1
   0  1  2  0
Right side of the table:
L(2)
   0
   1
   2
   0

Working with relation: (a:0 -> 1)(b:1 -> 2)(c:2 -> 0) = id
Here is the relation table (a:0 -> 1)(b:1 -> 2)(c:2 -> 0) = id after filling in consequences.
Table for relation: (a:0 -> 1)(b:1 -> 2)(c:2 -> 0) = id
Left side of the table:
L(0)  a  b  c
   0  1  2 -1
   1  2  0  1
   2  0  1  2
Right side of the table:
L(0)
   0
   1
   2

Working with relation: (b:1 -> 2)(c:2 -> 0)(a:0 -> 1) = id
Here is the relation table (b:1 -> 2)(c:2 -> 0)(a:0 -> 1) = id after filling in consequences.
Table for relation: (b:1 -> 2)(c:2 -> 0)(a:0 -> 1) = id
Left side of the table:
L(1)  b  c  a
   0  1  2  0
   1  2 -1 -1
   2  0  1  2
   0  1  2  0
Right side of the table:
L(1)
   0
   1
   2
   0
Exiting fillInConsequences() method.
Entering coincidencesExist() method.
Exiting coincidencesExist() method.
Entering dealWithCoincidenes() method.
Exiting dealWithCoincidenes() method.
Entering fillInConsequences() method.

Working with relation: (c:2 -> 0)(a:0 -> 1)(b:1 -> 2) = id

Here is the relation table (c:2 -> 0)(a:0 -> 1)(b:1 -> 2) = id after filling in consequences.

Table for relation: (c:2 -> 0)(a:0 -> 1)(b:1 -> 2) = id

Left side of the table:

L(2) | c a b
-----|-----|-----
 0   | 1  2  0
 1   | 2  0  1
 2   | -1 -1 -1
 0   | 1  2  0

Right side of the table:

L(2)

0
1
2
0

Working with relation: (a:0 -> 1)(b:1 -> 2)(c:2 -> 0) = id

Here is the relation table (a:0 -> 1)(b:1 -> 2)(c:2 -> 0) = id after filling in consequences.

Table for relation: (a:0 -> 1)(b:1 -> 2)(c:2 -> 0) = id

Left side of the table:

L(0) | a b c
-----|-----|-----
 0   | 1  2 -1
 1   | 2  0  1
 2   | 0  1  2

Right side of the table:

L(0)

0
1
2
0

Working with relation: (b:1 -> 2)(c:2 -> 0)(a:0 -> 1) = id

Here is the relation table (b:1 -> 2)(c:2 -> 0)(a:0 -> 1) = id after filling in consequences.

Table for relation: (b:1 -> 2)(c:2 -> 0)(a:0 -> 1) = id

Left side of the table:

L(1) | b c a
-----|-----|-----
 0   | 1  2  0
 1   | 2  -1 -1
 2   | 0  1  2
 0   | 1  2  0

Right side of the table:

L(1)

0
1
2
0

— Baker Mountain Proprietary —
Exiting fillInConsequences() method.
Entering coincidencesExist() method.
Exiting coincidencesExist() method.
Entering undefinedElements() method.
Exiting undefinedElements() method.
Entering defineNewElement() method.

Added element 3 to L(0) in L table for 2

L Table for 2 is now
L(2) c
   0 1
   1 2
   2 3
   0 1

L Table for 0 is now
L(0) a
   0 1
   1 2
   2 0
   3 -1

Exiting defineNewElement() method.
Entering fillInConsequences() method.

Working with relation: (c:2 -> 0)(a:0 -> 1)(b:1 -> 2) = id
Here is the relation table (c:2 -> 0)(a:0 -> 1)(b:1 -> 2) = id after filling in consequences.
Table for relation: (c:2 -> 0)(a:0 -> 1)(b:1 -> 2) = id
Left side of the table:
L(2) c a b
   0 1 2 0
   1 2 0 1
   2 3 -1 -1
   0 1 2 0
Right side of the table:
L(2)
   0
   1
   2
   0

Working with relation: (a:0 -> 1)(b:1 -> 2)(c:2 -> 0) = id
Here is the relation table (a:0 -> 1)(b:1 -> 2)(c:2 -> 0) = id after filling in consequences.
Table for relation: (a:0 -> 1)(b:1 -> 2)(c:2 -> 0) = id
Left side of the table:
L(0) a b c
   0 1 2 3
   1 2 0 1
   2 0 1 2
   3 -1 -1 -1
Right side of the table:
L(0)
   0
Working with relation: \((b:1 \rightarrow 2)(c:2 \rightarrow 0)(a:0 \rightarrow 1) = \text{id}\)
Here is the relation table \((b:1 \rightarrow 2)(c:2 \rightarrow 0)(a:0 \rightarrow 1) = \text{id}\) after filling in consequences.

Table for relation: \((b:1 \rightarrow 2)(c:2 \rightarrow 0)(a:0 \rightarrow 1) = \text{id}\)

Left side of the table:

<table>
<thead>
<tr>
<th></th>
<th>L(1)</th>
<th>b</th>
<th>c</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>-1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Right side of the table:

<table>
<thead>
<tr>
<th></th>
<th>L(1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Exiting `fillInConsequences()` method.
Entering `coincidencesExist()` method.
Added coincidence \((0, 0, 3)\) from relation table for relation \((a:0 \rightarrow 1)(b:1 \rightarrow 2)(c:2 \rightarrow 0) = \text{id}\)
Exiting `coincidencesExist()` method.
Entering `dealWithCoincidenes()` method.
Exiting `dealWithCoincidenes()` method.
Entering `fillInConsequences()` method.

Working with relation: \((c:2 \rightarrow 0)(a:0 \rightarrow 1)(b:1 \rightarrow 2) = \text{id}\)
Here is the relation table \((c:2 \rightarrow 0)(a:0 \rightarrow 1)(b:1 \rightarrow 2) = \text{id}\) after filling in consequences.

Table for relation: \((c:2 \rightarrow 0)(a:0 \rightarrow 1)(b:1 \rightarrow 2) = \text{id}\)

Left side of the table:

<table>
<thead>
<tr>
<th></th>
<th>L(2)</th>
<th>c</th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Right side of the table:

<table>
<thead>
<tr>
<th></th>
<th>L(2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Working with relation: \((a:0 \rightarrow 1)(b:1 \rightarrow 2)(c:2 \rightarrow 0) = \text{id}\)
Here is the relation table \((a:0 \rightarrow 1)(b:1 \rightarrow 2)(c:2 \rightarrow 0) = \text{id}\) after filling in consequences.

Table for relation: \((a:0 \rightarrow 1)(b:1 \rightarrow 2)(c:2 \rightarrow 0) = \text{id}\)

Left side of the table:

<table>
<thead>
<tr>
<th></th>
<th>L(0)</th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
Right side of the table:
L(0)
0
1
2
0

Working with relation: (b:1 -> 2)(c:2 -> 0)(a:0 -> 1) = id
Here is the relation table (b:1 -> 2)(c:2 -> 0)(a:0 -> 1) = id after filling in consequences.
Table for relation: (b:1 -> 2)(c:2 -> 0)(a:0 -> 1) = id

Left side of the table:
L(1)    b  c  a
0  1  2  0
1  2  0  1
2  0  1  2
0  1  2  0

Right side of the table:
L(1)
0
1
2
0

Exiting fillInConsequences() method.
Entering coincidencesExist() method.
Exiting coincidencesExist() method.
Entering dealWithCoincidenes() method.
Exiting dealWithCoincidenes() method.
Entering fillInConsequences() method.
Working with relation: (c:2 -> 0)(a:0 -> 1)(b:1 -> 2) = id
Here is the relation table (c:2 -> 0)(a:0 -> 1)(b:1 -> 2) = id after filling in consequences.
Table for relation: (c:2 -> 0)(a:0 -> 1)(b:1 -> 2) = id

Left side of the table:
L(2)    c  a  b
0  1  2  0
1  2  0  1
2  0  1  2
0  1  2  0

Right side of the table:
L(2)
0
1
2
0

Working with relation: (a:0 -> 1)(b:1 -> 2)(c:2 -> 0) = id
Here is the relation table (a:0 -> 1)(b:1 -> 2)(c:2 -> 0) = id after filling in consequences.
Table for relation: (a:0 -> 1)(b:1 -> 2)(c:2 -> 0) = id

Left side of the table:
L(0)    a  b  c
0  1  2  0
1  2  0  1

|0 1 2 0|
|1 2 0 1|
|2 0 1 2|
|0 1 2 0|

Exiting fillInConsequences() method.
Entering coincidencesExist() method.
Exiting coincidencesExist() method.
Entering dealWithCoincidenes() method.
Exiting dealWithCoincidenes() method.
Entering fillInConsequences() method.
Working with relation: (b:1 -> 2)(c:2 -> 0)(a:0 -> 1) = id
Here is the relation table (b:1 -> 2)(c:2 -> 0)(a:0 -> 1) = id after filling in consequences.
Table for relation: (b:1 -> 2)(c:2 -> 0)(a:0 -> 1) = id

Left side of the table:
L(0)    a  b  c
0  1  2  0
1  2  0  1

<p>|0 1 2 0|
|1 2 0 1|
|2 0 1 2|
|0 1 2 0|</p>
<table>
<thead>
<tr>
<th>b</th>
<th>c</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Right side of the table:
L(0)
0
1
2
0

Working with relation: \((b:1 \rightarrow 2)(c:2 \rightarrow 0)(a:0 \rightarrow 1) = \text{id}\)
Here is the relation table \((b:1 \rightarrow 2)(c:2 \rightarrow 0)(a:0 \rightarrow 1) = \text{id}\) after filling in consequences.
Table for relation: \((b:1 \rightarrow 2)(c:2 \rightarrow 0)(a:0 \rightarrow 1) = \text{id}\)
Left side of the table:
L(1) b c a
0 1 2 0
1 2 0 1
2 0 1 2
0 1 2 0

Right side of the table:
L(1)
0
1
2
0

Exiting fillInConsequences() method.
Entering coincidencesExist() method.
Exiting coincidencesExist() method.
Entering undefinedElements() method.
Exiting undefinedElements() method.
Entering cleanup() method.
Left Kan extension

Name : Example 9 on page 156 of Bob Walters’ book
Objects : 3
Generators:

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

Relations :
\[(a:0 \rightarrow 1)(b:1 \rightarrow 2)(c:2 \rightarrow 0) = \text{id}\]
\[(b:1 \rightarrow 2)(c:2 \rightarrow 0)(a:0 \rightarrow 1) = \text{id}\]
\[(c:2 \rightarrow 0)(a:0 \rightarrow 1)(b:1 \rightarrow 2) = \text{id}\]

Action definition

Objects

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

Morphisms

<table>
<thead>
<tr>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
</tr>
</tbody>
</table>

\[(0, 1)(1, 2)(2, 0)\]
b: 1 -> 2
   (0, 1)(1, 2)(2, 0)

a: 0 -> 1
   (0, 1)(1, 2)(2, 0)
Appendix C

Haskell Implementations of Particular Kan Extensions

C.1 QKan.Classical.RightKanExtensions

--- Module : QKan.Classical.RightKanExtensions
--- Author : Ralph L. Wojtowicz (ralphw@bakermountain.org)
--- Version : 1 December 2012

module QKan.Classical.RightKanExtensions
(Function,
 UFUnclion,
 compose,
 terminalObject,
 product',
 equalizer,
 pullback,
) where

type Function = ([Int], Int)

type UFUnclion = Int -> Function

-- Any singleton set 1 is a terminal object in the category of finite sets
-- (or the category of sets). This object satisfies a universal mapping
-- property: given any set X, there is a unique function X -> 1.
terminalObject :: (Int, UFUnclion)
terminalObject = (1, shriek)
  where shriek = (\n -> ((take n (repeat 1)), 1))

-- A product of two sets X and Y consists of a set X x Y and two functions
-- pi0 : X x Y -> X and pi1 : X x Y -> Y that satisfy a universal mapping
-- property. One solution is given by cartesian product. In the case of
-- skeletal sets (i.e., sets of the form {0, 1, ..., n-1}), multiplication
-- gives the product operation.

product' :: Int -> Int -> (Int, (Function, Function))
product' x y = (xy, (pi0, pi1))
   where xy = x*y
       pi0 = ([div m y | m <- [0..(xy-1)]], x)
       pi1 = ([mod k y | k <- [0..(xy-1)]], y)

-- An equalizer of two functions f:X->Y and g:X->Y is a function
-- e:X'-> X that satisfies a universal mapping property. One solution
-- is given by X' = {x in X | f(x)=g(x)}.

equalizer :: Function -> Function -> Function
equalizer f g = (e, x)
   where x = length (fst f) -- the domain of f and g
       y = snd f
       f' = fst f
       g' = fst g
       e = [i | i <- [0..(x-1)], (f' !! i) == (g' !! i)]

-- Composition of functions f:X->Y and g:Y->Z.
compose :: Function -> Function -> Function
compose g f = (gf, z)
   where x = length(fst f)
       z = snd g
       gf = [(fst g) !! fx | fx <- fst f]

-- Implementation of a pullback of f:X->Z and g:Y->Z. We use
-- the construction of pullbacks from a product and an equalizer.
pullback :: Function -> Function -> (Function, Function)
pullback f g = (f', g')
   where x = length (fst f)
       y = length (fst g)
       p = product' x y
       pi0 = fst (snd p) -- the projection X x Y -> X
       pi1 = snd (snd p) -- the projection X x Y -> Y
       fpi0 = compose f pi0
       gpi1 = compose g pi1
       e = equalizer fpi0 gpi1
       f' = compose pi0 e
       g' = compose pi1 e
C.2 QKan.Classical.LeftKanExtensions

-- Module : QKan.Classical.LeftKanExtensions
-- Author : Ralph L. Wojtowicz (ralphw@bakermountain.org)
-- Version : 12 September 2012

module QKan.Classical.LeftKanExtensions

( Function,
  UFunction,
  compose,
  initialObject,
  coproduct',
  coequalizer,
) where

import Data.List

type Function = ([Int], Int)

type UFunction = Int -> Function

-- Composition of functions f:X->Y and g:Y->Z.
compose :: Function -> Function -> Function
compose g f = (gf, z)
  where x = length(fst f)
        z = snd g
        gf = [(fst g) !! fx | fx <- fst f]

-- The empty set 0 is an initial object in the category of finite sets
-- (or the category of sets). This object satisfies a universal mapping
-- property: given any set X, there is a unique function 0 -> X.
initialObject :: (Int, UFunction)
initialObject = (0, coshriek)
  where coshriek = (\n -> ([], n))

-- A coproduct of two sets X and Y consists of a set X x Y and two functions
-- i0 : X -> X + Y and i1 : Y -> X + Y that satisfy a universal mapping
-- property. One solution is given by disjoint union. In the case of
-- skeletal sets (i.e., sets of the form {0, 1, ..., n-1}), addition
-- gives the product operation.
coproduct' :: Int -> Int -> (Int, (Function, Function))
coproduct' x y = (x_plus_y, (i0, i1))
  where x_plus_y = x + y
        i0 = (\m -> [m | m <- [0..(x-1)]], x_plus_y)
i1 = ([n+x | n <- [0..(y-1)]], x_plus_y)

-- A coequalizer of two functions f : X -> Y and g : X -> Y is a function
-- q : Y -> Q that satisfies a universal mapping property. One solution
-- has Q = Y/~ where ~ is the equivalence relation on Y generated pairs
-- (f(x), g(x)) where x ranges over x.
coequalizer :: Function -> Function -> Function
coequalizer f g = (q, y')
  where y = snd f -- the codomain of f and g
        pairs = zip (fst f) (fst g) -- edges in a graph with y as vertex set
        startIndices = [0..(y-1)]
        q = shuffle $ foldl convertByIndex startIndices pairs
        y' = length $ nub q

-- Change the value y to x if x == x', otherwise, return y. In C, the
-- swap function could be programmed as swap(x,x')(y) = (x == x') ? x : y
swap :: (Integral a) => a -> a -> a -> a
swap x x' y
  | y == x' = x
  | otherwise = y

-- If x <= x', then convert [x0, ..., xn] (x, x') changes each occurrence
-- of x in the list to x'. Otherwise, it converts each occurrence of x' to x.
convert :: (Integral a) => [a] -> (a, a) -> [a]
convert xs (x,x')
  | x <= x' = map (swap x x') xs
  | otherwise = map (swap x' x ) xs

-- Similar to convert but the input pair (i,j) refers to indices in the list.
convertByIndex :: [Int] -> (Int, Int) -> [Int]
convertByIndex xs (i,j) = convert xs (x,x')
  where x = xs!!i
        x' = xs!!j

-- Ensure that the integers occurring in a list would be [0, ..., n] after sorting.
-- For example, [0, 1, 3] would be converted to [0, 1, 2].
shuffle :: [Int] -> [Int]
shuffle xs = foldl convert xs pairs
  where g = sort $ nub xs
        f = [0..((length g) - 1)]
        pairs = zip f g
Appendix D

Development and Documentation

This document, our reports, presentations and software were all produced with freely-available tools. These tools include the Debian Linux operating environment (www.debian.org) and the \LaTeX{} document preparation system (www.tug.org). For software development we used the Java (www.java.sun.com) and Haskell (www.haskell.org) programming languages, the emacs text editor and make (www.gnu.org/software). Please support the use of free software.
Bibliography


Index

Ban$_\infty$, 79
\[ \square, 4 \]
$C^\circ$ (dual category), 9, 83
$\Delta$, iii, 63, 90, 97, 101, 115, 116, 119, 120
$\mathbb{N}$, xvi
$O$, xvi, 91, 123
$\Omega$, xvi, 101
$\mathbb{R}$, xvi
$\mathbb{R}^+$, xvi
$\Theta$, xvi, 92
$\mathbb{Z}$, xvi
$\!, 30, 31, 79, 115, 119, 183$
$i, 39, 79, 119, 185$
$\sim, 14$
$\varepsilon$, 151
$\approx, 22$
$\sim, 14$
$\{ | \}, 4$
o, xvi
$\sqcup$, 4
$\mathcal{P}$, 154
$\bigcup$, 153

Ab, 83
abelian
  category, iii, 79, 83, 84
  group, iii, 1, 65, 78, 79, 82, 83
Abelian Stabilizer Problem, 70, 78
action, 59, 87, 109
action-schema.xsd, 109
addition, i–iii, 1, 92, 101, 120, 123
carry-lookahead, 90, 92
carry-save, 90
ripple-carry, 92
additive category, 83
adjoint, 1, 36, 48, 67, 73
Adjoint Functor Theorem, 44
adjunction, 45
count, 46
unit, 46
algorithm
  Carmody-Walters, xiii, 87
  Deutsch, xiii
  Deutsch-Jozsa, xiii
  Knuth-Bendix, xiii
  Simon, xiii
  Todd-Coxeter, xiii, 87, 95
Apache commons, iv, 103
Applied Communication Sciences, 122
array multiplier, 90
asymptotic
  lower bound, xvi
  upper bound, xvi
asymptotically
  tight bound, xvi
balanced, 12
Banach space, 79
basis, 17
BBN, iv, v
benchmark, iv, 157–161, 182
bijection, 59
bijective function, 10
biproduct, 82, 83
boxed subroutine, 124
Carmody-Walters Algorithm, i, xiii, 65, 87, 89, 113, 122, 125
carry-lookahead addition, 90, 92
carry-save addition, 90
cartesian
  category, xv, 60
  product, 155
cartesian product, 60, 90, 97
category, 3, 6–8, 67
  abelian, 79, 83, 84
  additive, 83

197
balanced, 12
cartesian, xv, 60
cocomplete, 41, 43
comma, 25, 42, 43, 55
complete, xv, 36, 37, 60
dual, 9, 11, 83
finitely-presented, 5
locally small, 4, 82
preadditive, 82
proto-, 8
simplicial, 63
small, 3, 5
sub-, 9
theory of, 5, 7, 8
characteristic function, 91, 99, 100
Church-Turing Thesis, 70
circuit transformation, 124
class, 4
claw, iii, 91, 99, 118
claws, i
cocomplete category, 41, 43
cicone, 38
   isomorphism, 39
   morphism, 38
codomain, 4
coequalizer, iii, iv, 1, 40, 41, 61, 62, 83, 89, 92, 97, 100, 101, 115, 120, 123, 125, 185
cokernel, 81–84
colimit, 39, 41, 43, 97, 115
   coequalizer, 40
   coproduct, 40
   coterminator, 39
   pushout, 40
collision, 99
comma category, 25, 42, 43, 55
commutative diagram, xv
commute, xv
compilation, 125
complete
   category, xv, 60
complete category, 36, 37
complex matrix, iv, 103
composite
   functor, 15
   natural transformation, 19
composition, 3, 118
Comprehension
   Axiom of, 153
cone, 30
   isomorphism, 30
   morphism, 30
   conservative, 17
   constant
      functor, 14
   constant on
      left cosets, 72
      right cosets, 72
   contact information, v
   continuous functor, 48, 73
   coproduct, i, iii, iv, 1, 40, 61, 82, 83, 97, 101, 115, 120, 123, 125, 185
coset, xiii, 66
   constant function on, 72
   enumeration, 87, 95
   representative of, 66
coterminator, 39, 61
   counit
      of an adjunction, 46
Dalhousie, iv
Dalhousie University, 122
data, 34
deliverables, xiv
Deutsch’s algorithm, xiii
Deutsch’s Problem, 76
Deutsch-Jozsa algorithm, xiii
Deutsch-Jozsa Problem, 76
diagonal functor, 23
diagram
   commutative, xv
discrete iterator, 94
Discrete Logarithm Problem, 78
disjoint union, 92, 100
disjoint-set forest, 93
documentation, iv
domain, 4
   of a path, 5
dominating set, iii, 1, 93, 101
Dominating Set Problem, i
dual
   category, 9, 83
   functor, 15
natural transformation, 18
dual category, 11
dynamic lifting, 124
dynamic system, 1, 10, 36
Eclipse IDE, 124
emacs, 187
embedding
  Yoneda, 29
empty word, 67
epimorphism, 10, 11, 48, 80, 83, 84
equality
  of functions, 59
  of functors, 15
  of natural transformations, 18
equalizer, i, iii, iv, 1, 36, 60, 81, 83, 91, 97, 99, 100, 115, 117, 118, 123, 125, 183
  limit, 33
equation, 32
equivalence
  of categories, 22, 47
  relation, 40, 41, 61, 62, 89, 92, 97, 100, 101, 120, 122
evaluation functor, 23
Extent
  Axiom of, 151, 154
factor group, 78
factoring, 79
faithful, 15
finite
  set, 63
finitely-presented category, 5
fixed point, 10
forgetful functor, 14
Foundation
  Axiom of, 155
foundations, 4
Fourier transform, iii, 79, 122, 123
free group, 1, 67, 68
Freyd, Peter, 44
full
  functor, 16
  subcategory, 9
Function, 115–120, 183, 185
function, 10
  bijective, 59
equality, 59
injective, 59
surjective, 59
functor, 13, 65, 87, 97, 115
  composition, 15
  conservative, 17
  constant, 14
  continuous, 48, 73
diagonal, 23
dual, 15
embedding, 15
evaluation, 23
  faithful, 15
  forgetful, 14
full, 16
hom, 14, 26, 29
identity, 13
inclusion, 14
reflective, 43
representable, 42
underlying object, 14, 65, 67, 72–74
generating
  set of objects, 17
generator, 17
generators, 95
ghci, 116, 119
ghci, 116, 119
goals, xiii
Group, 67
group, i, 65, 79, 82, 83, 95
  abelian, 65, 78
  as a category, 65
  factor, 78
  free, 67, 68
  homomorphism, 67
  presentation, 67, 68
  representation, 95
  solvable, 78
  symmetric, 66, 72
group action, 70
Grover’s Algorithm, i
Grover’s Search Algorithm, 1, 99, 100, 123
haddock, iv
Haselgrove, 68
Haskell, iv, 115–120, 122, 124, 125, 183, 185, 187
Heyting algebra, ii
Hidden Subgroup Problem, i, iii, 1, 73, 76, 77
hom
  functor, 14, 26, 29
homomorphism, 67, 71, 82
  as a functor, 65
homotopy, iv
IARPA, iv, 115
identity
  element, 65
  functor, 13
  natural transformation, 18
identity morphism, 3
image, 62
inclusion, 14
index of a subgroup, 66, 68, 87
Infinity
  Axiom of, 153
initial object, iv, 39, 79, 82, 125, 185
injection, 59
intersection, 155
invariant manifold, 10
invariant subobject, 36
inverse, 65
isomorphism, 11
  in Set, 60
  natural, 21
iterator, 94
Java, iv, 103, 122, 124, 125, 187
javadoc, iv, 103, 111
JAXB, iv, 103
Kan
  lifting, iv
Kan extension, ii, xiii, 1, 87, 89, 91, 92
  computational example, 89
left, 55, 87
  as a right, 55
  colimit formula, 55
right, 48
  as a left adjoint, 53
  limit formula, 49
kernel, 81–84
kernel pair, 36
Knuth-Bendix Procedure, xiii
\LaTeX, 187
lattice, ii
lazy evaluation, 122, 124, 125
left
  Kan extension (see Kan extension), 55
left coset, 66
length of a word, 67
lifting
  Kan, iv
limit, 10, 30, 36, 43, 91, 97, 115
  equalizer, 32, 33
  kernel pair, 36
  product, 31
  pullback, 34
  terminator, 30
Linux, 124, 187
locale, ii
locally small category, 4, 82
logic, 1
lower bound, xvi
Mac OSX, 124
make, 187
membership, 10, 151
module, 79, 83
monad, 1
monic, 10, 33, 35
  regular, 33
monomorphism, 10, 11, 35, 48, 80, 81, 83, 84
  in Set, 59
  in Set^J, 28
monomorphism, 10
morphism, 3, 10
multiplication, i–iii, 1, 90, 97, 98, 116, 123
  array, 90
  Wallace-tree, 90
natural isomorphism, 21
natural map, 71
natural monomorphism, 28
natural transformation, 18
  component, 18
  composition, 19
  dual, 18
  horizontal composition, 20
  identity, 18
  product, 20
vertical composition, 19
normal series, 78
normal subgroup, 67, 68, 71

NP-complete, iii, 1

object, 3
generating, 17
initial, iv, 39, 79, 82, 185
separating, 17
terminal, iv, 30, 79, 82, 183
zero, 79, 81
on-the-fly circuit generation, 122, 124
one-to-one function, 10
onto function, 10
orbit, 71
ordered pair, 155

pair
ordered, 155
partition, 66
path, 5
path compression, 93
Period-Finding Problem, 78
point, 31
pointed set, 79
Power Set
Axiom, 152
power set, 154
preadditive category, 82
presentation
category, 95
group, 95
presentation, of a group, 67, 68
product, i, iii, iv, 1, 31, 36, 60, 82, 83, 97, 99, 115, 116, 118, 123, 125, 183
cartesian, 90, 97, 155
in Set, 90
proto-category, 8
pullback, i, iii, iv, 1, 35, 60, 91, 98–100, 118, 125, 183
pushout, 40
QuaFL, iv, v, 115, 124
quantifier, 1
quantum counting, 123
Quick-Union, 93
Quipper, iv, v, 115, 122

reflect
epimorphisms, 16
isomorphisms, 17
monomorphisms, 16
reflection, 41–43
reflective functor, 43
regular
monic, 33
relation, 5
relations, 95
Replacement
Axiom of, 153
representation, 42
group, 95
representative of a coset, 66
right coset, 66
right Kan extension, 48
ring, 79, 83
ripple-carry addition, 92

schema
XML, 103, 109
separating
set of objects, 17
separator, 17
Set, ii, 59, 79, 89
set
finite, 63, 90, 97, 101, 115, 116, 119, 120
power, 154
theory, 4, 59
Setf, ii, iii, 63, 90, 97, 101, 115, 116, 119, 120
Shor’s Order Finding Problem, 77
σ-algebra, 94
Simon’s algorithm, xiii
Simon’s Problem, 77
simplicial category, ii, 63, 90, 97, 101, 115, 116, 119, 120
small category, 3, 5
solution, 32, 34
Solution Set Condition, 44
solvable group, 78
source, 4
source code, iv
source-target predicate, 9
stabilizer, 71
subcategory, 9
full, 9

subgroup, 66, 71
  index of, 66, 68
  normal, 67, 68

subobject, 59, 81

subset, 152

surjection, 59

symmetric group, 66, 72

synthesis, 125

target, 4

terminal object, iv, 30, 79, 82, 115, 119, 125, 183

terminator, 30, 60

\(\mathrm{TeX}\), 187

theory
  of categories, 5, 7, 8
  of proto-categories, 9

tight bound, xvi

Todd-Coxeter Procedure, i, iv, xiii, 65, 68, 70, 87, 89, 95, 113, 122, 125

topology, 94

TORQUE, 124

transformation, 18
  component, 18
  natural, 18

Triangle Finding Algorithm, 123

Turing machine, 70

\(\mathrm{UFunction}\), 115, 119

\(\mathrm{UFunction}\), 183, 185

underlying object functor, 14, 65, 67, 72–74

underlying set, 65

union, 92, 100, 154
  disjoint, 61

Union Axiom, 153

Union-Find, 93

unit
  of an adjunction, 46

universal
  arrow, 41
  element, 42

University of Pennsylvania, 122

upper bound, xvi

Wallace-tree multiplier, 90

Windows, 124

word, 67

Word Problem, 70

XML, iv, 103, 113, 157–161, 182
  schema, 103, 109

Yoneda
  embedding, 29
  lemma, 26

Zermelo-Fraenkel axioms, 59

zero morphism, 79

zero object, 79, 81–83

Zorn’s Lemma, iii, 73
Dr. Ralph L. Wojtowicz  
Baker Mountain Research Corporation  
P. O. Box 68  
Yellow Spring, WV 26865-0068

Dr. Noson S. Yanofsky  
Department of Computer  
and Information Science  
Brooklyn College  
2900 Bedford Avenue  
Brooklyn, NY 11210