Understandable Proofs in Graphical Calculi

Michael Wang
Worcester College
University of Oxford

A thesis submitted for the degree of
Master of Mathematics and Computer Science

Trinity 2019

Acknowledgements

I thank my supervisors, Bob and Dan, for their great advice, for proposing this fascinating investigation, and for welcoming me into the community. I also thank the other wonderful people in the ZX-calculus group at Oxford, especially Hector and Miriam, for their encouragement.

I furthermore thank my tutors, Michael, Andrzej, Hongseok, Robin, Rich, David, Geraint, Alessandro, and Stephen for their innumerable hours of teaching, support, and encouragement over the course of my degree.

I give thanks to my other teachers over the course of my degree: the lecturers, class tutors and practical demonstrators who worked hard to ensure we properly understood the material.

I also give thanks to all my teachers at school, who stimulated and inspired me to get to this point.

I thank the staff at Worcester College for their support and help, and for helping me navigate the labyrinth of university organisation and life.

Finally, I thank my friends in Oxford.

Abstract

A new layout algorithm for graphical proofs, *rewrite rule-based drawing*, is developed. It aims to produce understandable proofs in graphical calculi that use double-pushout (DPO) rewriting. Although interactive theorem provers have been successful at ensuring the correctness of proofs, proof readability and understandability is often damaged in comparison to conventional mathematical practice. This work resolves this issue by developing, practically implementing, and evaluating new layout algorithms, directed by the characteristics of effective layouts. The Quantomatic proof assistant's internal layout architecture is also refactored for extensibility and correctness. This culminates in a novel graphical proof layout system that is efficient, extensible, and practical, and produces semantically meaningful output.

Contents

Chapter 1

Introduction

How do you ensure that a correct proof is *understandable*?

Through the computer-based technology of *interactive theorem proving*, the opposite has been achieved: well-understood proofs can be verified as formally correct. Such tools are now commonly used by researchers in theoretical computer science and formalist mathematics. However, computers have no care for human readability preferences, or even for producing proofs that are practical for humans to read and understand. This combination has led to the production of unreadable proofs.[1]

The problems with interactive theorem proving and other computer-aided proof tools have resulted in the vast majority of mathematicians continuing to work in human language, where proofs can be easily written and understood. The better understood the proof, the easier it is to verify, whether by human or computer. Indeed, mathematics is so well-understood that the 'social process of mathematics' [DeMillo, Lipton, and Perlis, 1979] is sufficient to make conjectures, prove theorems, and verify the work of others, with extraordinarily high accuracy. Even when there are errors, the intuition of mathematicians is surprisingly accurate: prior to the publication of the collected papers of David Hilbert [Rota, 1997], Olga Taussky-Todd spent three years reading through, understanding the material, and correcting the mistakes. In the end, only one mistake was found to be uncorrectable: Hilbert's proof of the continuum hypothesis. Today, the continuum hypothesis is known to be, in

[1]A particularly egregious example, largely generated by computer, is [Garvie and Duncan, 2017]. Another example is [Duncan and Lucas, 2013, appendix A].

fact, independent of the axioms of Zermelo-Fraenkel set theory [Cohen, 1963].

Thus, *understandable* theorem proving should be a high priority for future computer-aided proof tools. In this thesis, I present two novel algorithms to automate understandable proof layout for *graphical calculi*, a software implementation of the most effective algorithm in the Quantomatic interactive theorem prover, and evaluate the effectiveness of the system in practice. Graphical calculi are (essentially) rewriting systems for graphs; rewriting typically uses the double-pushout (DPO) algorithm (although other rewriting algorithms exist). Due to their visual nature, graphs are significantly more intuitive than terms of formal languages, which have traditionally dominated research in formalist mathematics and computer-aided proof tools. Examples of these calculi are the ZX-calculus [Coecke and Kissinger, 2017, Backens, 2016], ZW-calculus [Hadzihasanovic, 2015], and ZH-calculus [Backens and Kissinger, 2018].

1.1 Related Work

Human-readable proof display in proof assistance and automation tools is largely underexplored, although some do recognise it as a barrier to mainstream adoption of proof assistance and automation tools [Bundy, 2011]. However, the [NuPRL] proof assistant is able to generate hypertext where the reader may examine the proof and its structure by selecting the subgoals that interest them, in the order that they choose. [Matita] also provides hypertext which enables definitions to be looked up easily, and also provides mathematical typesetting through MathML [Asperti et al., 2007]. Many popular tools, such as [Coq] and [Isabelle], support incremental proof verification, which allows readers to watch tactics generate, reduce, and eliminate proof subgoals. Other tools, such as [Mizar] provide syntax highlighting and/or code folding.

The field of graph layout is quite large: many books have been written about it, and there is even an annual conference dedicated to the topic, the *International Symposium on Graph Drawing*. Many software packages for graph layout, such as [Graphviz], are distributed on the Internet; they usually provide multiple algorithms. The idea to provide multiple algorithms is good, because this facilitates the communication of disparate concepts.

However, typical off-the-shelf software packages are designed to present a single static graph; they are not intended to present the evolution of one graph into another using rewriting rules, and therefore they create sudden changes between consecutive graphs. This issue of 'dynamic graph drawing' is considered in [Branke, 2001], in the context of operations which either add or remove vertices and/or edges, and primarily discusses how existing algorithms can be adapted to avoid sudden, distracting layout changes. However, the most important problem with existing graph layout work is that there is little consideration of string and spider diagrams, which have a characteristic structure distinct from graphs that are random or arise from applied network theory; the latter are the focus of most research in undirected graph layout, such as [Kaufmann and Wagner (eds.), 2001]. In comparison, my research specifically concerns the dynamic layout of graphical proofs.

The development of the [Quantomatic] theorem prover and its underlying theory is well-documented in the literature, particularly in past Doctor of Philosophy theses from Oxford. The theory of graphical calculi for quantum information and the initial development of Quantomatic are explained in detail in [Kissinger, 2012]. A more recent, but technically outdated (in that the software architecture of Quantomatic has, since, changed) is [Kissinger and Zamdzhiev, 2015]. The theory that allows reasoning on string diagrams of categories (with sufficient structure) to be treated as reasoning on graphs is introduced in [Dixon and Duncan, 2009] for compact closed categories, and then is given a categorical description in [Dixon, Duncan, and Kissinger, 2010] which [Dixon and Kissinger, 2013] connects with the structure of adhesive categories. The theory of rewriting on !-graphs, which allows graph equations to be treated as universally quantified on natural numbers of particular subgraphs, is developed in [Merry, 2013]. Layout-wise, Quantomatic, prior to this thesis, supported calling [Graphviz] to run the 'dot' algorithm (which needs to be separately installed), and implemented a layout algorithm for single graphs and single (incremental) rewrites that is similar to finding equilibrium of physical forces using the forward Euler numerical method. (The latter technique is referred to as 'force-directed' in the literature [Hu, 2006].)

1.2 Contributions

The novel contributions of this thesis are:

- Two novel algorithms for graphical proof layout are presented.
- Time and space complexity of the algorithms are considered.
- The most practical of the algorithms is implemented in the *Quantomatic* theorem prover.
- In the *Quantomatic* theorem prover, the internal software architecture of the layout components is revamped to be cleaner and more compositional.
- I evaluate the performance and quality of the layout algorithms' output, using the software implementation.

1.3 Outline

The remainder of my thesis is structured as follows.

In chapter 2, I summarise the background material, including graph theory and graphical calculi, more formally.

In chapter 3, I define the scope and goals of proof layout algorithms for graphical calculi, and then describe two novel algorithms. I also evaluate the theoretical time and space complexity.

In chapter 4, I describe the software architecture and implementation of my revamp to the layout components in the Quantomatic proof assistant, which implements the automatic diagram layout system described above. I also describe graphical user interface changes.

In chapter 5, I evaluate the diagram layout system that I have designed using experimental data: I benchmark the software implementation for efficiency, and more importantly, I verify whether humans prefer the output of the new diagram layout system.

In chapter 6, I summarise my novel developments and methodology, evaluate the effectiveness, and suggest possibilities for future work.

Chapter 2

Preliminaries

I first introduce the preliminary mathematical theory, in order to introduce the foundational concepts, and specify the particular definitions that are used in this thesis. Elementary knowledge about numbers and sets is assumed.

2.1 Graph theory

A *digraph*, or *directed graph*, is a quadruple of a set of *vertices* V, a set of *edges* E, a function $\mathit{source} : E \to V$, and a function $\mathit{target} : E \to V$. A *digraph* can equivalently be defined as a pair (V, E), where V is a set of *vertices* and E is a multiset containing elements from $V \times V$, representing the *edges* of the graph by pairs of their source and target vertices. What is referred to in this thesis as a digraph may be known in other literature as a *quiver*, *multidigraph*, or *directed multigraph*.

An (undirected) *graph* is a triple of a set of *vertices* V, a set of *edges* E, and a function $\mathit{ends} : E \to \{\{u, v\} \mid u, v \in V\}$. An (undirected) *graph* can equivalently be defined as a pair (V, E), where V is a set of *vertices* and E is a multiset containing elements from $\{\{u, v\} \mid u, v \in V\}$, representing the *edges* of the graph by their end vertices. (Note that edges from a vertex to itself may occur in the graph; in other words, both $u = v$ and $u \neq v$ may occur inside the set comprehensions in this paragraph.) What is referred to in this thesis as an (undirected) graph may be known in other literature as an *(undirected) multigraph*.

The advantage of the edge-and-functions definition (the former in the two paragraphs above) over the edge-multiset definition (the latter in the two paragraphs above) is that the former allows each edge to be distinguished; each element of E, in the edge-and-functions definition, represents the *identity* of an edge (in the philosophical and object-oriented programming sense). Additional values can be associated with each edge by defining new functions with domain E. The analogous construction in the edge-multiset definition is more complicated. (Adding information to each edge in a graph may be known in other literature as *edge labelling.*) (Of course, vertices can also have associated values; this may be known in other literature as *vertex labelling*).

In other literature, it may be common to abbreviate, in the definition of a digraph, the functions *source* and *target* to s and t, respectively. Examples include [Dixon and Duncan, 2009], [Dixon, Duncan, and Kissinger, 2010], and [Dixon and Kissinger, 2013]. A detailed source for these definitions follows [Derksen and Weyman, 2005], which refers to *quivers*, *heads*, and *tails* for what (in this thesis) are called *directed graphs*, *sources*, and *targets*. Quiver theory and category theory (mentioned in the next section) also have many connections, although they will not be discussed in this thesis.

In much literature, the restriction is imposed that each pair of vertices can have either zero or one edge (in a directed graph) from the first vertex to the second, or (in an undirected graph) between them. This restriction is not imposed here. In this thesis, adding this restriction creates (respectively, directed or undirected) *unigraphs.* Unlabelled multigraphs can be viewed as unigraphs where edges are labelled with nonzero natural numbers corresponding to the number of edges in the multigraph represented by the single edge in the unigraph.

A *finite* (directed or undirected) graph is one where V and E are finite sets.

Any undirected graph can be considered as a directed graph $(V, E, \mathit{source}, \mathit{target})$ where there exists a bijection $f : E \to E$ such that for every edge $e \in E$, $\mathit{source}(e) = \mathit{target}(f(e))$ and $\mathit{target}(e) = \mathit{source}(f(e))$.

A homomorphism between directed graphs $(V, E, \mathit{source}, \mathit{target})$ and $(V', E', \mathit{source}', \mathit{target}')$ is a pair of functions $f : V \to V'$ and $g : E \to E'$ such that for every $e \in E$, $\mathit{source}'(g(e)) = f(\mathit{source}(e))$ and $\mathit{target}'(g(e)) = f(\mathit{target}(e))$. Similarly, a homomorphism between undi-

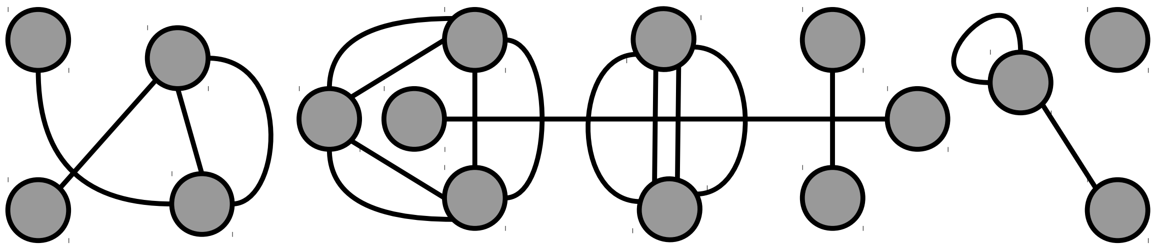

Figure 2.1: Example of a graph and a possible drawing in two dimensions

rected graphs (V, E, ends) and (V', E', ends') is a pair of functions $f : V \to V'$ and $g : E \to E'$ such that for every $e \in E$, $\mathit{ends}'(g(e)) = f[\mathit{ends}(e)]$. (On labelled graphs, the conditions on the functions are extended to the label functions similarly, in that they should commute with f or g as appropriate.) In both directed and undirected cases, if f and g have inverses $f^{-1} : V' \to V$ and $g^{-1} : E' \to E$ which form a homomorphism in the reverse direction, then the homomorphism is an *isomorphism*.

Graphs can be visualised in n-dimensional (real) space, $\mathbb{R}^n$, by *drawing*. Graphs are often drawn in two or three dimensions, because typical readers are familiar with these dimensionalities. One-dimensional graph drawings are almost always ambiguous, so this dimensionality is only used in unambiguous situations (e.g. for graphs with up to two vertices) or where context resolves the ambiguity (e.g. in timelines). Figure 2.1 is an example of a two-dimensional drawing of an undirected graph.

2.2 Graph rewriting

There are many definitions of graph rewriting. The *double-pushout* definition is used for the definition of graphical calculi based on string diagrams [Kissinger, 2012]. It is typically given for directed graphs, but is general for adhesive categories [Lack and Sobociński, 2004] (which, conversely, can be seen as abstracting the graph-like structures which support rewriting). Here, I give a definition based on [Lack and Sobociński, 2004] and [Kissinger, 2012, §4.3], but attempting to avoid (or make optional) the terminology of category theory. For brevity, I also avoid diagrams, but [Kissinger, 2012, §4.2] has excellent diagrams and is highly recommended. When the word "graph" is used in this section, it can be read to mean a directed or undirected graph, labelled or unlabelled, as long as it is consistently read in the same way.

A *rewrite rule* (or *production* in the parlance of [Lack and Sobociński, 2004]) is a triple of three graphs K, L, and R, where K is a subgraph of both L and R. Abstractly, this subgraph constraint can be expressed as requiring two injective graph homomorphisms $K \to L$ and $K \to R$ (which should be seen as inclusion homomorphisms). Intuitively, L and R are the left and right sides of the rewrite rule, while K is a subgraph common to L and R which provides a 'boundary' for the rewrite rule and acts as an 'interface' to the rest of the graph. The $K \to L$ homomorphism is often called l, and the $K \to R$ homomorphism is often called r.

A set of rewrite rules defines a *graphical calculus* (or *grammar* in the parlance of [Lack and Sobociński, 2004]).

To *rewrite* a graph G into a graph H using a rewrite rule (K, L, R, l, r), we require an injective graph homomorphism from L to G, $(m, n) : (V_L \to V_G) \times (E_L \to E_G)$, such that for every vertex $v \in V_L \quad V_K$, every edge $e \in E_G$ that satisfies $\mathit{target}(e) = m(v)$ (or, for undirected graphs, $m(v) \in \mathit{ends}(e)$) is in the image of the function n [Kissinger, 2012, theorem 4.3.13]. Intuitively, L embeds into G, and touches its surroundings using only the vertices from the graph K. Form I, which is K extended with the surroundings of G and H, and a subgraph of G and H. Alternatively, I is formed by removing the vertices and edges in G that come from L and are not also from K. Then, the vertices and edges that are in R (that do not come from K) are added, connecting up with the interface K found in I, to form H. Abstractly, there are injective graph homomorphisms from K into I, and from I into G and H.

Graph rewriting enables *diagrammatic reasoning* on drawn graphs, and is the primary mechanism for proof in Quantomatic. A *derivation* in Quantomatic is a series of applications of graph rewriting.

2.3 Spider calculi

For a detailed treatment of this topic, see [Coecke and Kissinger, 2017].

Spider calculi are a special case of string diagram calculi. The boxes in spider calculi are *spiders*, which are families of boxes (morphisms), with a box for each number of input and output wires (which are all of the same wire type (system type/object)). There may be different kinds of spiders. Spiders may be decorated with *phases.* All spider families satisfy the following rules:

- is equal to itself composed with arbitrary permutations of the order of inputs and the order of outputs
- is equal to another spider of the same kind when inputs are bent to outputs or vice versa
- merge with spiders of the same kind which are connected by at least one wire

Spider calculi may also satisfy additional axioms. These axioms primarily describe how spiders of different kinds interact. *Complementarity* and *strong complementarity* are important axioms in ZX-calculus, where the Z spider and X spider represent different, strongly complementary bases (of linear algebra). A single spider calculus cannot have more than 2 kinds of spiders which are pairwise strongly complementary [Coecke and Kissinger, 2017, theorem 9.65].

Spider calculi can be viewed mathematically as † special commutative Frobenius algebras on a vector space. [Coecke and Kissinger, 2017, proposition 8.101] Strongly complementary spider calculi are then recognisable as Hopf algebras [Coecke and Kissinger, 2017, §9.6.1].

Chapter 3

Algorithm Design

3.1 Scope

The aim is to develop a proof layout algorithm for the Quantomatic proof assistant which lays out proofs so that they are understandable. Quantomatic introduces these additional considerations:

- It is not possible to control the geometry of the rendered wires beyond adding additional wire vertices. By default, for each pair of vertices, the edges between them are drawn as curves centred densely around the straight line between the vertices.

  Overall, this is a benefit. For layout purposes, all multigraphs can be considered as unigraphs, as multiple edges between the same pair of vertices are treated the same when rendered to an image. Furthermore, this display is ideal from a semantic perspective, as such edges are not inherently distinguishable and therefore have similar meaning.

- Quantomatic supports *!-boxes*, which can contain vertices and edges, and semantically represent a subgraph which can be duplicated any number of times or deleted, and enable Quantomatic to prove more general theorems. Edges can cross from outside to inside a !-box, and they are duplicated or deleted with the !-box.

  Layout of !-boxes needs extra care. Contents of !-boxes should be laid out together; however, laying them out too close could introduce ambiguity and spacing issues).

This project primarily focuses on two-dimensional drawing, because this is what is supported by Quantomatic's architecture. However, the novel algorithms later will be presented generally for any dimensionality.

3.2 Goals

The algorithm should lay out graphs in a way that visually establishes semantically important concepts, such as symmetry, common substructures, and the stasis of parts not involved in a rewrite step.

Symmetry is a recurring concept in mathematics with an effective visual representation. Many formulae are described as symmetric. For example $x^2 + xy + y^2$ (in $\mathbb{R}$) is symmetric in the variables x and y, because swapping x and y results in an equivalent formula. Symmetry in diagrammatic and graphical notations is more striking, invoking the intuitive notion of geometric symmetry to represent semantic symmetry. Thus, in order to ease understanding, diagrams and diagram equations with semantic symmetry should ideally be laid out with syntactic, visual symmetry.

Diagrams, especially equations between diagrams, will contain *common and recurring substructures.* These common substructures form familiar 'units' with particular meanings and which become rewritten in similar ways. Therefore, these common substructures should ideally receive a consistent layout throughout any proof, and across different proofs, including while they are being rewritten to related structures. This facilitates a common understanding of the structure, both within a proof and between different proofs.

For proofs where rules are applied to subgraphs of some graph in order to rewrite them, it is crucial that *only rewritten graph elements move* and that *non-rewritten graph elements are static.* In particular, this is the case for double-pushout graph rewriting, where the left-hand and right-hand sides of a rule are literally included inside any of its applications. Semantically, it is the inclusion of the (left-hand side of the) rule inside the larger graph which is being rewritten. Therefore, syntactically, this inclusion should receive a new layout, while the layout of the parts which are not being rewritten should remain static across the rewriting step.

3.3 Brute-force drawing

There is a simple, brute-force algorithm, using dynamic programming to maintain important characteristics: symmetry, common layouts between substructures, and changing only the layout of rewritten subgraphs.

Given a derivation, we can collect every possible connected subgraph in each graph in the derivation, up to isomorphism (where graphs with different labels are not isomorphic), into a set. Then, the principle of dynamic programming can be applied. Given each subgraph, from smallest to largest, compute a layout with optimal symmetry. Then, rewrite all the larger subgraphs, replacing the current subgraph with a special node representing the current subgraph. The output graph layouts (for each step of the derivation) are then obtained by replacing (by graph rewriting) the special nodes with the corresponding laid-out subgraphs.

The advantage of this algorithm is its simplicity, and hence that it can easily be verified to generate graphs which have the characteristics above. The algorithm clearly respects substructures of diagrams by design, and does so across the entire derivation. It can also be seen that the reuse of subgraph layouts across the entire derivation causes only the rewritten parts in each step to change layout.

The primary problem with this algorithm is its time and space complexity. The number of connected subgraphs is, in general, superpolynomial [Eppstein, 2013]. Furthermore, the graph isomorphism problem (which decides whether two graphs are isomorphic) is not known to be solvable in polynomial (or less) time [Schöning, 1987]. (Readers are advised not to confuse this with the subgraph isomorphism problem, which decides for two graphs whether the first has a subgraph isomorphic to the second. Set data structures typically are tree-based or table-based, and therefore the relevant time complexity is that of the graph isomorphism problem.) Therefore, the worst-case time and space complexity of this drawing algorithm is superpolynomial.

3.4 Rewrite rule-based drawing

I now suggest an algorithm based on the structure of the graph rewriting process. In particular, this algorithm is modelled on the operation of double-pushout (DPO) rewriting, as described in §2.2, although the technique generalises to other rule-based graph rewriting algorithms. This algorithm can operate individually on each proof step, and can also operate on a whole proof in a simple, compositional manner.

The core idea is to lay out each rewriting step by incorporating a well-known layout of the rule into the inclusions of the rule in the graph being rewritten. The well-known layouts are typically designed or identified by people. Often, one person will design the well-known layout and then share it with others, as already happens with the Quantomatic sample project library. It is assumed that these well-known layouts are semantically meaningful and understandable to human readers. Although there is human labour involved in specifying such well-known rule layouts, it needs only be done once for each rule, and is much easier than having to specify the layouts of entire proofs (which would necessarily involve more steps involving larger graphs).

For each step where a rule is applied, the algorithm aims to incorporate a well-known layout for that rule in the 'best' way. This is done by finding the affine transformation that minimises the sum of squared vertex distances between the image of the rule's left-hand side under the affine transformation and the inclusion of the rule's left-hand side in the left-hand side of the proof step. This affine transformation is then applied to the rule's right-hand (and, optionally, the left-hand) side, and then these new coordinates are used to lay out the inclusion of the rule's right-hand (and left-hand) sides in the proof step. Informally, the algorithm finds the affine transformation that best 'lines up' the rule's left-hand side and its inclusion in the proof step's left-hand side, and then applies the algorithm in order to fully 'line up' the rule and its inclusion in the proof step.

Formally, let V be the (nonempty) set of vertex names in the left-hand side of a rule, $(\mathbf{w}_v)_{v\in V}$ represent the coordinate of vertex v in the rule's left-hand side, and $(\mathbf{z}_v)_{v\in V}$ represent the coordinate of the inclusion of vertex v in the proof step's left-hand side (representing all 2D coordinates directly as 2D column vectors).

A general n-dimensional affine transformation has the form $g(\mathbf{x}) = \mathbf{A}\mathbf{x}+\mathbf{b}$ for some $n\times n$ matrix $\mathbf{A}$ and an n-dimensional column vector $\mathbf{b}$. A 2-dimensional affine transformation therefore has the form $g(\mathbf{x}) = \left(\begin{smallmatrix} a & b \\ c & d \end{smallmatrix}\right)\mathbf{x} + \left(\begin{smallmatrix} e \\ f \end{smallmatrix}\right)$ for $a, b, c, d, e, f \in \mathbb{R}$.

Lagrangian optimisation is now used to find the matrix $\mathbf{A}$ and vector $\mathbf{b}$ that minimises the objective function $L(\mathbf{A}, \mathbf{b}) = \sum_{v\in V} |\mathbf{z}_v - (\mathbf{A}\mathbf{w}_v + \mathbf{b})|^2$, which is the sum of squared Euclidean distances between corresponding vertices. The partial derivatives of L with respect to each entry of the matrix $\mathbf{A}$ and the vector $\mathbf{b}$ are:

$$\begin{aligned}
\frac{\partial L}{\partial(\mathbf{A})_{ij}} &= \frac{\partial}{\partial(\mathbf{A})_{ij}} \sum_{v\in V} |\mathbf{z}_v - (\mathbf{A}\mathbf{w}_v + \mathbf{b})|^2 \\
&= \frac{\partial}{\partial(\mathbf{A})_{ij}} \sum_{v\in V} \sum_{m=1}^{n} \left((\mathbf{z}_v)_m - \left(\sum_{k=1}^{n} (\mathbf{A})_{mk} (\mathbf{w}_v)_k \right) - (\mathbf{b})_m \right)^2 \\
&= \sum_{v\in V} \sum_{m=1}^{n} \frac{\partial}{\partial(\mathbf{A})_{ij}} \left((\mathbf{z}_v)_m - \left(\sum_{k=1}^{n} (\mathbf{A})_{mk} (\mathbf{w}_v)_k \right) - (\mathbf{b})_m \right)^2 && \text{(sum rule)} \\
&= \sum_{v\in V} \frac{\partial}{\partial(\mathbf{A})_{ij}} \left((\mathbf{z}_v)_i - \left(\sum_{k=1}^{n} (\mathbf{A})_{ik} (\mathbf{w}_v)_k \right) - (\mathbf{b})_i \right)^2 && \text{(eliminate zero terms)} \\
&= \sum_{v\in V} -2(\mathbf{w}_v)_j \left((\mathbf{z}_v)_i - \left(\sum_{k=1}^{n} (\mathbf{A})_{ik} (\mathbf{w}_v)_k \right) - (\mathbf{b})_i \right) && \text{(chain rule)} \\
&= -2 \sum_{v\in V} \left((\mathbf{w}_v)_j (\mathbf{z}_v)_i - \left(\sum_{k=1}^{n} (\mathbf{w}_v)_j (\mathbf{A})_{ik} (\mathbf{w}_v)_k \right) - (\mathbf{w}_v)_j (\mathbf{b})_i \right) && \text{(distributivity)} \\
&= -2 \left(\left(\sum_{v\in V} (\mathbf{w}_v)_j (\mathbf{z}_v)_i \right) - \left(\sum_{k=1}^{n} (\mathbf{A})_{ik} \sum_{v\in V} (\mathbf{w}_v)_j (\mathbf{w}_v)_k \right) - \left(\sum_{v\in V} (\mathbf{w}_v)_j \right) (\mathbf{b})_i \right)
\end{aligned}$$

$$\begin{aligned}
\frac{\partial L}{\partial(\mathbf{b})_{i}} &= \frac{\partial}{\partial(\mathbf{b})_{i}} \sum_{v\in V} |\mathbf{z}_v - (\mathbf{A}\mathbf{w}_v + \mathbf{b})|^2 \\
&= \sum_{v\in V} \sum_{m=1}^{n} \frac{\partial}{\partial(\mathbf{b})_{i}} \left((\mathbf{z}_v)_m - \left(\sum_{k=1}^{n} (\mathbf{A})_{mk} (\mathbf{w}_v)_k \right) - (\mathbf{b})_m \right)^2 && \text{(as before)} \\
&= \sum_{v\in V} \frac{\partial}{\partial(\mathbf{b})_{i}} \left((\mathbf{z}_v)_i - \left(\sum_{k=1}^{n} (\mathbf{A})_{ik} (\mathbf{w}_v)_k \right) - (\mathbf{b})_i \right)^2 && \text{(eliminate zero terms)} \\
&= \sum_{v\in V} -2 \left((\mathbf{z}_v)_i - \left(\sum_{k=1}^{n} (\mathbf{A})_{ik} (\mathbf{w}_v)_k \right) - (\mathbf{b})_i \right) && \text{(chain rule)} \\
&= -2 \left(\left(\sum_{v\in V} (\mathbf{z}_v)_i \right) - \left(\sum_{k=1}^{n} (\mathbf{A})_{ik} \sum_{v\in V} (\mathbf{w}_v)_k \right) - (\mathbf{b})_i |V| \right) && \text{(distributivity)}
\end{aligned}$$

Setting each partial derivative to 0 results in a system of $n(n+1)$ simultaneous linear equations with $n(n+1)$ variables. However, by inspecting the structure of the simultaneous equations more closely, the system can be separated into n systems of $n+1$ simultaneous linear equations (with each system having $n+1$ variables). This is achieved because for each $i \in [1..n]$, $\frac{\partial L}{\partial (\mathbf{A})_{ij}}$ and $\frac{\partial L}{\partial (\mathbf{b})_i}$ vary only with the $n+1$ variables $(\mathbf{A})_{ik}$ for each $k \in [1..n]$ and $(\mathbf{b})_i$, and do not vary with any other parameters of L. For each $i \in [1..n]$, each system of $n+1$ simultaneous equations can then be represented as an equation of a symmetric matrix (which is the same for any $i \in [1..n]$), a vector of constant terms (which does depend on $i \in [1..n]$), and a vector of the system's variables: (Lines have been added to clarify the matrix structure.)

$$\left(\begin{array}{cccc|c}
\sum\limits_{v\in V}(\mathbf{w}_v)_1(\mathbf{w}_v)_1 & \sum\limits_{v\in V}(\mathbf{w}_v)_1(\mathbf{w}_v)_2 & \cdots & \sum\limits_{v\in V}(\mathbf{w}_v)_1(\mathbf{w}_v)_n & \sum\limits_{v\in V}(\mathbf{w}_v)_1 \\
\sum\limits_{v\in V}(\mathbf{w}_v)_2(\mathbf{w}_v)_1 & \sum\limits_{v\in V}(\mathbf{w}_v)_2(\mathbf{w}_v)_2 & \cdots & \sum\limits_{v\in V}(\mathbf{w}_v)_2(\mathbf{w}_v)_n & \sum\limits_{v\in V}(\mathbf{w}_v)_2 \\
\vdots & \vdots & \ddots & \vdots & \vdots \\
\sum\limits_{v\in V}(\mathbf{w}_v)_n(\mathbf{w}_v)_1 & \sum\limits_{v\in V}(\mathbf{w}_v)_n(\mathbf{w}_v)_2 & \cdots & \sum\limits_{v\in V}(\mathbf{w}_v)_n(\mathbf{w}_v)_n & \sum\limits_{v\in V}(\mathbf{w}_v)_n \\
\hline
\sum\limits_{v\in V}(\mathbf{w}_v)_1 & \sum\limits_{v\in V}(\mathbf{w}_v)_2 & \cdots & \sum\limits_{v\in V}(\mathbf{w}_v)_n & |V|
\end{array}\right)
\left(\begin{array}{c}
(\mathbf{A})_{i1} \\ (\mathbf{A})_{i2} \\ \vdots \\ (\mathbf{A})_{in} \\ \hline (\mathbf{b})_i
\end{array}\right)
=
\left(\begin{array}{c}
\sum\limits_{v\in V}(\mathbf{w}_v)_1(\mathbf{z}_v)_i \\
\sum\limits_{v\in V}(\mathbf{w}_v)_2(\mathbf{z}_v)_i \\
\vdots \\
\sum\limits_{v\in V}(\mathbf{w}_v)_n(\mathbf{z}_v)_i \\
\hline
\sum\limits_{v\in V}(\mathbf{z}_v)_i
\end{array}\right)$$

The systems of simultaneous linear equations can then be solved for all variables by any suitable method, e.g. Gaussian elimination or matrix inversion. The optimal solution of the variables then give the optimal matrix $\mathbf{A}$ and the optimal vector $\mathbf{b}$. Note that Lagrangian optimisation will never produce a maximum in this case, because the objective function L has no maximum: for any $x \in \mathbb{R}$, take $\mathbf{A} = \left(\begin{smallmatrix}1 & 0\\ 0 & 1\end{smallmatrix}\right)$ and $\mathbf{b} = \mathbf{z}_u - \mathbf{w}_u - \left(\begin{smallmatrix}\sqrt{|x|+1}\\ 0\end{smallmatrix}\right)$ for some arbitrarily chosen $u \in V$. Then, for this particular $u \in V$, $|\mathbf{z}_u - \mathbf{A}\mathbf{w}_u - \mathbf{b}|^2 = |\mathbf{z}_u - \mathbf{w}_u - \mathbf{z}_u + \mathbf{w}_u + \left(\begin{smallmatrix}\sqrt{|x|+1}\\ 0\end{smallmatrix}\right)|^2 = |\left(\begin{smallmatrix}\sqrt{|x|+1}\\ 0\end{smallmatrix}\right)|^2 = \sqrt{|x|+1}^2 + 0^2 = |x|+1 > |x| \geq x$. As every summand of $\sum_{v\in V} |\mathbf{z}_v - (\mathbf{A}\mathbf{w}_v + \mathbf{b})|^2$ is non-negative, $L(\mathbf{A}, \mathbf{b}) = \sum_{v\in V} |\mathbf{z}_v - (\mathbf{A}\mathbf{w}_v + \mathbf{b})|^2 \geq |\mathbf{z}_u - \mathbf{A}\mathbf{w}_u - \mathbf{b}|^2 > x$. Therefore, L has no maximum.

For the two-dimensional case, which is used in Quantomatic, there are 2 systems of

$2 + 1 = 3$ simultaneous equations, where system i is represented by the matrix equation:

$$\begin{pmatrix} \sum_{v\in V}(\mathbf{w}_v)_1(\mathbf{w}_v)_1 & \sum_{v\in V}(\mathbf{w}_v)_1(\mathbf{w}_v)_2 & \sum_{v\in V}(\mathbf{w}_v)_1 \\ \sum_{v\in V}(\mathbf{w}_v)_2(\mathbf{w}_v)_1 & \sum_{v\in V}(\mathbf{w}_v)_2(\mathbf{w}_v)_2 & \sum_{v\in V}(\mathbf{w}_v)_2 \\ \sum_{v\in V}(\mathbf{w}_v)_1 & \sum_{v\in V}(\mathbf{w}_v)_2 & |V| \end{pmatrix} \begin{pmatrix} (\mathbf{A})_{i1} \\ (\mathbf{A})_{i2} \\ (\mathbf{b})_i \end{pmatrix} = \begin{pmatrix} \sum_{v\in V}(\mathbf{w}_v)_1(\mathbf{z}_v)_i \\ \sum_{v\in V}(\mathbf{w}_v)_2(\mathbf{z}_v)_i \\ \sum_{v\in V}(\mathbf{z}_v)_i \end{pmatrix}$$

Unfortunately, simultaneous equations do not always have a unique solution, or even a solution at all. The algorithm resolves this in 2D by successively trying more restricted types of affine transformation until one provides a unique solution. First, it is well known that (in 2D) three vertices are required to define an affine transformation, so the general case of affine transformations need not be tested if $|V| < 3$. If $|V| < 3$ or no unique solution is found, the algorithm then attempts to find the optimal scale, rotation, and translation (an affine transformation of form $g(\mathbf{x}) = \left(\begin{smallmatrix} a & b \\ -b & a \end{smallmatrix}\right)\mathbf{x} + \left(\begin{smallmatrix} c \\ d \end{smallmatrix}\right)$ for $a, b, c, d \in \mathbb{R}$), thus attempting to solve the matrix equation shown in figure 3.1. If this also fails, the algorithm then attempts to find the optimal scale and translation (a transformation of form $g(\mathbf{x}) = a\mathbf{x} + \left(\begin{smallmatrix} b \\ c \end{smallmatrix}\right)$ for $a, b, c \in \mathbb{R}$), solving a similar matrix equation. If this also fails, then the algorithm resorts to using a simple translation of the rule's LHS. This always succeeds, as the algorithm simply computes the mean deviation between each $\mathbf{z}_v$ and $\mathbf{w}_v$.

It is possible for a rule to have multiple well-known layouts, and the most appropriate of these would be selected algorithmically (by selecting the well-known layout for which, under its optimal affine transformation, had the least sum of squared vertex distances, as already shown) for each proof step. However, this technique could not be implemented in Quantomatic (for reasons explained in §4.1), although important groundwork for such an implementation was laid.

The affine transformation so derived is then applied to each vertex coordinate of both the left-hand and right-hand graphs in the proof step's rule, in order to compute a new layout for the inclusion of both those graphs in the proof step. When laying out a whole proof, the new layout of the left-hand graph in each proof step is discarded, except for the root graph. (This is because the layout has nowhere else to go, although this could easily

$$\begin{pmatrix} \left(\sum_{v\in V}(\mathbf{w}_v)_1(\mathbf{w}_v)_1\right)+\left(\sum_{v\in V}(\mathbf{w}_v)_2(\mathbf{w}_v)_2\right) & 0 & \sum_{v\in V}(\mathbf{w}_v)_1 & \sum_{v\in V}(\mathbf{w}_v)_2 \\ 0 & \left(\sum_{v\in V}(\mathbf{w}_v)_1(\mathbf{w}_v)_1\right)+\left(\sum_{v\in V}(\mathbf{w}_v)_2(\mathbf{w}_v)_2\right) & \sum_{v\in V}(\mathbf{w}_v)_2 & -\sum_{v\in V}(\mathbf{w}_v)_1 \\ \sum_{v\in V}(\mathbf{w}_v)_1 & \sum_{v\in V}(\mathbf{w}_v)_2 & |V| & 0 \\ \sum_{v\in V}(\mathbf{w}_v)_2 & -\sum_{v\in V}(\mathbf{w}_v)_1 & 0 & |V| \end{pmatrix} \begin{pmatrix} a \\ b \\ c \\ d \end{pmatrix} = \begin{pmatrix} \left(\sum_{v\in V}(\mathbf{w}_v)_1(\mathbf{z}_v)_1\right)+\left(\sum_{v\in V}(\mathbf{w}_v)_2(\mathbf{z}_v)_2\right) \\ \left(\sum_{v\in V}(\mathbf{w}_v)_2(\mathbf{z}_v)_1\right)-\left(\sum_{v\in V}(\mathbf{w}_v)_1(\mathbf{z}_v)_2\right) \\ \sum_{v\in V}(\mathbf{z}_v)_1 \\ \sum_{v\in V}(\mathbf{z}_v)_2 \end{pmatrix}$$

Figure 3.1: The matrix equation to solve when finding the optimal affine transformation that is only a scale, rotation, and translation.

be changed by adding reflexivity (a.k.a. no-op) proof steps.)

To illustrate the core operation of the algorithm, the output of the algorithm when laying out the step `disconnect-0` in the derivation `n-disconnect` from the Quantomatic sample project `zh`, which applies the axiom `disconnect` (from the same project), is shown in figure 3.2. First, the algorithm reads the proof step, and its current left-hand side. Then, the rule is loaded, and the algorithm finds the most optimal affine transformation to transform from the rule's left-hand into the proof step's left-hand inclusion of the rule (highlighted in red). This transformation is then applied to the inclusions of the rule on both sides (highlighted in red and blue on the left-hand and right-hand sides respectively), thus creating a new layout for the proof step based on the rule.

In terms of the goals of graphical proof layout, rewrite rule-based drawing obviously maintains the stasis of non-rewritten graph elements, and only changes the layout of rewritten subgraphs. Similarly, common substructures inherently share a similar (sub-)layout, due to the well-known layouts of each rule's left-hand and right-hand side being included into the proof step. (Smaller substructures are also maintained if the well-known layouts also have common layouts for common substructures.) However, symmetry is only achieved if the well-known layouts for the rewrite rules have symmetry. Fortunately, it is almost always the case (for the Quantomatic sample projects) that well-known rule layouts have symmetry.

The time complexity, in terms of the number of vertices in the rule's left-hand $|V|$ and the number of vertices in the rule's right-hand $|R|$, is $\Theta(|V| + |R|)$. (This holds both for a single proof step, and for a whole proof.) This is justified thus. Computing various sums over the coordinates of the vertices (with no nesting below the top level of summation) is $\Theta(|V|)$. These sums then participate in matrix operations which are constant-time with respect to $|V|$ (since their size and number depends only on the dimensionality). Applying the computed affine transformation then occurs for each vertex on both sides of the rule, and is $\Theta(|V| + |R|)$. Adding these time complexities gives the result.

Taking the dimensionality of the drawing space n into account, the asymptotic time complexity of the algorithm is $O((|V| + |R|)n^2 + n^{3.373\ldots})$.[1] This will be seen thus.

[1]If the conjecture that matrix inversion is $O(n^2)$ holds, then the total asymptotic time complexity is

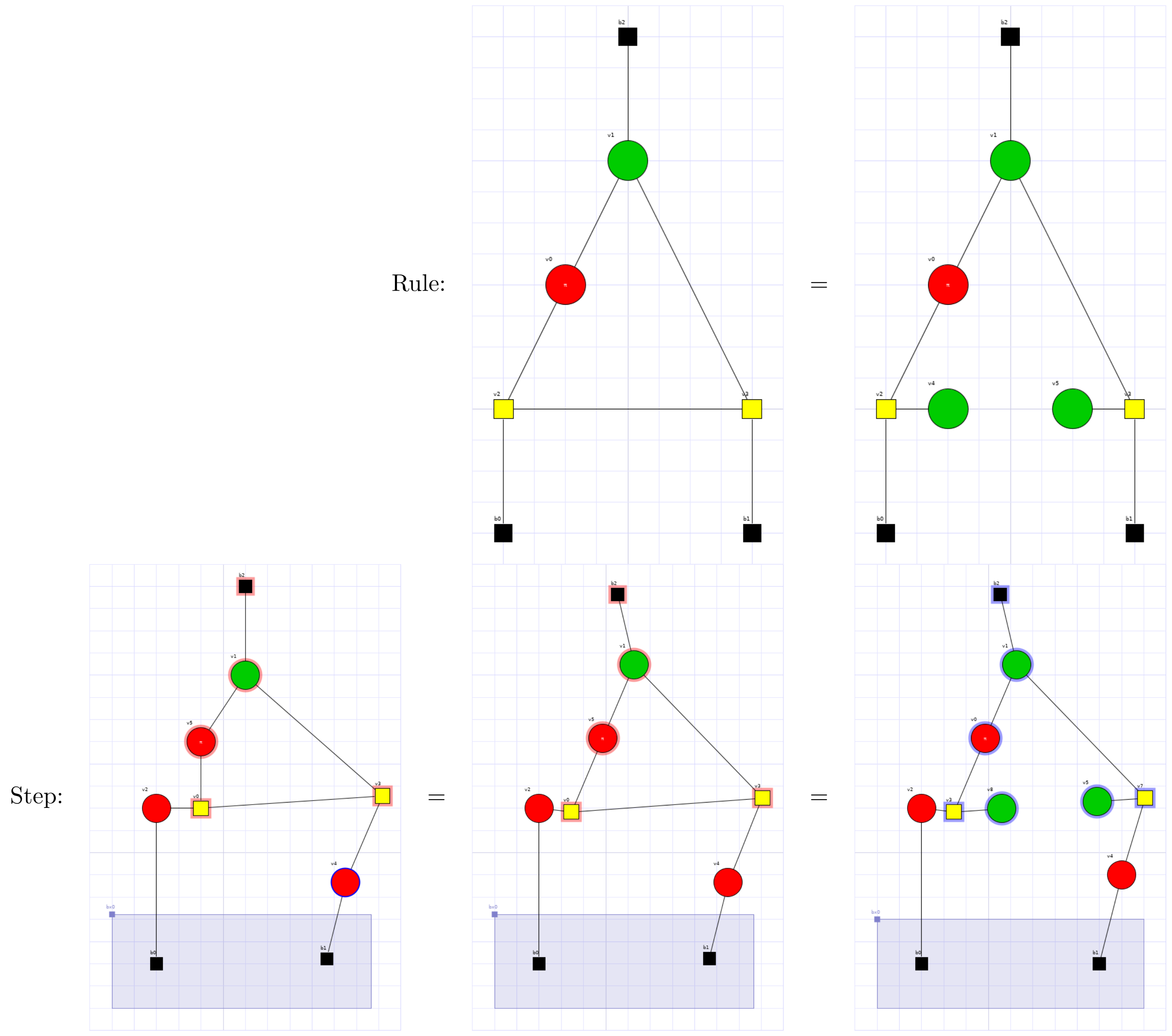

Figure 3.2: Example of applying rule-based drawing to step `disconnect-0` in derivation `n-disconnect`, which uses the rule `disconnect` (an axiom). Notice how both the left-hand and the right-hand sides of the derivation step are laid out to match the layout of the rule.

First notice that the matrix used to calculate the optimal affine transformation is the same for each system. Therefore, the matrix only needs to be computed once, and its inverse also only needs to be computed once. Each entry of the matrix is a single sum over the vertices, except for the bottom-right entry. Taking advantage of the matrix's symmetry, $\frac{n(n+1)}{2} + n = \frac{n(n+3)}{2}$ entries need to be computed. Since each sum takes $\Theta(|V|)$ time, the overall computation of the matrix has time complexity $\Theta(\frac{n(n+3)|V|}{2}) = \Theta(|V|n^2)$. The vectors (on the right-hand of the matrix equations, of which there are n) also need to be computed. Each of the vectors' $n+1$ entries contains a sum, so computing the vectors also has time complexity $\Theta(n(n+1)|V|) = \Theta(|V|n^2)$.

Matrix inversion has asymptotic time complexity at least $O(m^2)$ (where $m \times m$ is the dimensionality of the square matrix): the best known algorithm, Coppersmith-Winograd-Stothers, has time complexity $O(n^{2.373\ldots})$ [Davie and Stothers, 2013] [Williams, 2014]. Other algorithms have higher asymptotic time complexity (but possibly smaller constant factors). Thus, the splitting of a system of $n(n+1)$ linear equations into n systems of $n+1$ linear equations gives a performance boost, from $O((n(n+1))^{2.373\ldots}) = O(n^{4.746\ldots})$ (where the system is not split) to $O((n+1)^{2.373\ldots}n) = O(n^{3.373\ldots})$ (where the system is split into n systems).[2]

Finally, computing the new layouts is done for each vertex, and evaluating a general affine transformation has time complexity $\Theta(n^2 + n) = \Theta(n^2)$. Since there are $|V| + |R|$ vertices to lay out, this final step has time complexity $\Theta((|V| + |R|)n^2)$.

The asymptotic space complexity of this algorithm is $\Theta(n^2)$, because the only data that is stored is the inverse of the $(n+1) \times (n+1)$ matrix, a[3] vector of size $n+1$, and the affine transformation (which is $(n+1) \times (n+2)$ values).

$O((|V| + |R|)n^2 + n^3)$

[2]If the conjecture that matrix inversion is $O(n^2)$ holds, then the improvement in asymptotic time complexity gained from splitting the linear system as described is from $O((n(n+1))^2) = O(n^4)$ to $O((n+1)^2 n) = O(n^3)$.

[3]The storage used for each of the n vectors can be reused for the next vector.

Chapter 4

Software Architecture and Implementation

I designed and implemented a new architecture for layout of all document types supported by Quantomatic, which is written in the Scala programming language. Using this framework, I then cleanly implemented rewrite rule-based drawing. I also enhanced the graphical user interface to support the new layout functionality.

4.1 Existing document architecture

Quantomatic's document types (representing the top-level user-editable objects) are `Graph`, `Rule`, and `Derivation`. In broad strokes, a `Rule` comprises both a left-hand and a right-hand `Graph`. A `Derivation` comprises a root `Graph`, a set of named `DSteps`[1], and a partial map of `DStep` names to `DStep` names, representing the parent of each `DStep` in the `Derivation`. A `DStep` comprises a `Rule` and an output (right-hand) `Graph` to the parent `DStep`. The source (left-hand) graph for a `DStep` is the output of its parent `DStep`, or, if there is no parent, the root `Graph` of the `Derivation` tree. In Quantomatic, data objects, including documents, are immutable; modification occurs by assigning new data to reassignable variables outside of the data objects, such as in the user interface's model.

Objects inside a document are represented using *name* values and mappings from names

[1]'derivation step'

to object properties. This is similar to the Entity-Component pattern [Martin, 2007]. For example, `Derivation` names its `DStep`s using `DSName`s, which are the Entities, and its two Components are the map from `DSName`s to `DStep`s and the parent-step map from `DSName`s to `DSName`s. A more sophisticated example is `Graph`, which contains vertices, edges, and !-boxes. These are the kinds of Entity, while the data about them is represented in Components: mappings of vertex names to `VData` (a triple of coordinate, vertex type, and annotation), of edge names to `EData` (a triple of directedness, edge type and annotation), from !-box names to `BBData` (merely an annotation), from !-box names to parent !-box names, and two mappings, called `source` and `target`, from edge names to vertex names. Interestingly, these last two mappings replicate the edge-and-functions definition from §2.1. Indeed, the edge-and-functions definition of a graph can be viewed as an application of the Entity-Component pattern in mathematics.

Unfortunately, the fact that vertices are bound to a single coordinate each in the `VData` type means that each `Graph` stores exactly one layout. Multiple layouts for a single `Graph` are not possible. As mentioned in §3.4, this prevents support for multiple well-known layouts for a single `Rule`. The `GraphLayoutData` interface, introduced in the next section §4.2, acts as a container of layout data (and as a Component of a `Graph`) and therefore can become the foundations of a future improvement to Quantomatic to support multiple layouts for a single `Graph`.

4.2 Revamped layout architecture

I designed a revamped architecture for computing layouts. Each of the types `Graph`, `Rule`, `DStep`, and `Derivation` is associated with a `LayoutStrategy` interface, instantiating the Strategy design pattern, which provides a common interface to different algorithms [Gamma et al., 1994]. Implementations of each `LayoutStrategy` interface may provide additional configuration methods and constructors. Each `LayoutStrategy` interface outputs values of a `Layout` interface, corresponding to the document type and representing the layout data for that document's content. To simplify the design, `Rule` and `DStep`, each composed of two `Graph`s, simply output in the form of two `GraphLayoutData` instances.

Document type	`Graph`	`Rule`
LayoutStrategy interface	`GraphLayoutStrategy`	`RuleLayoutStrategy`
Layout interface	`GraphLayoutData`	—
Document type	`DStep`	`Derivation`
LayoutStrategy interface	`DStepLayoutStrategy`	`DerivationLayoutStrategy`
Layout interface	—	`DerivationLayout`

Table 4.1: Associations between each document class and its layout interfaces.

The specific names of the interfaces for each document type are shown in table 4.1, and the interfaces' definitions are listed in appendix A. (The `GraphLayoutData` interface would be named simply `GraphLayout`, but that conflicts with the name of the legacy `GraphLayout` mixin, which was previously used for `Graph` layout. Also, each `DStepLayoutStrategy` operates on both a `DStep` and the source `Graph` that it rewrites.) Particular layout algorithms are thus expressed by implementing the `LayoutStrategy` and `Layout` interfaces for the respective document type. Since `Layout` is merely an interface, implementations of algorithms need not generate a large table of vertex coordinates, enabling deferred computation and computational and memory savings.

The overall design follows the mantra of software architecture, *"Program to an 'interface', not an 'implementation'."*, from [Gamma et al., 1994]. Furthermore, I deliberately kept the interfaces as simple as practicable, in order to enable the widest possible range of implementations.

Unfortunately, the legacy layout algorithms were implemented with a problematic interface, `GraphLayout`. Its scope is too maximal, implementing (in the interface) 'vertex locking', which is handled poorly by most, if not all, algorithms implemented using the legacy interface. (A code inspection suggests that 'vertex locking' was designed purely to support the stasis of non-rewritten graph elements, as described in §3.2. This intention is better solved by the more general `DStepLayoutStrategy`, which allows the layout algorithm to explicitly keep non-rewritten graph elements static, encouraging implementations to properly support the behaviour. Implementations of graph layout algorithms which *do* support 'vertex locking' are also supported within the new design, which explicitly accommodates algorithm-specific parameterisation.) By default, the legacy interface also randomises the coordinate data before executing the layout algorithm. This behaviour should also be left to individual layout algorithms, since some layout algorithms may simply be designed to enhance an existing layout, like filters in image editing programs, where randomisation-by-default is inappropriate. Furthermore, `GraphLayout` is implemented by mutating a `Graph` instance, which is then returned to the caller. This makes it possible for a buggy implementation of the `GraphLayout` interface to corrupt graph data, e.g. by adding or removing vertices or edges. This is particularly dangerous for a proof assistant, as the implementations of `GraphLayout` become part of (and enlarge) the proof assistant's trusted computing base. Finally, the original `GraphLayout` interface was completely undocumented.

Nonetheless, the large quantity of work on the legacy layout algorithms must be acknowledged. I have documented every public attribute and method in the `GraphLayout` interface, and additionally made several efficiency improvements, such as eliminating repeated tests of a variable which remains constant. Furthermore, I implemented the Adapter design pattern [Gamma et al., 1994], so that `GraphLayout` implementations can be adapted to the `GraphLayoutStrategy` interface using the `GraphLayoutAdapterGraphLayoutStrategy` class. This makes previously-developed `Graph` layout algorithms available to newer code. Furthermore, the use of this this adapter class moves `GraphLayout` implementations out of the trusted computing base (since the `GraphLayoutData` interface is designed to prevent implementations from altering `Graph` data other than the coordinates of vertices). Finally, I updated the pre-existing derivation layout implementation, which was merely a class `DeriveLayout` (which provided a totally fixed implementation of a single layout algorithm, force-based layout with clustering), to implement the `DerivationLayoutStrategy` and `DerivationLayout` interfaces, as classes `ForceClusterDerivationLayoutStrategy` and `ForceClusterDerivationLayout`.

Most importantly, I implemented the rewrite rule-based drawing algorithm described in §3.4. The implementation exists as the classes `RuleDStepLayoutStrategy` and `RuleDStepLayout`, implementing the `DStepLayoutStrategy` and `DStepLayout` interfaces respectively. The implementation has some additional optimisations for cases where the number of vertices in the left-hand side of the rule are 0 or 1: this often is the case for rules made up mostly of boundary vertices, which inherently disappear in Quantomatic's rewriting system (as *wires*[2] are matched up to homeomorphism, and often normalised to single edges) [Kissinger, 2012]. (I also updated Graph.expandWire so that the new wire vertex that it inserts is at a defined, sensible location: namely, the midpoint of the edge that the wire vertex splits.) In the 0-vertices case, there is simply no transformation, and no transformation is applied to either side, leaving the vertices as-is. In the 1-vertex case, the affine transformation is immediately resolved to a pure translation. To perform the matrix inversion, the [Breeze] library was used, which claims to be efficient. Matching Quantomatic's existing numerical precision, all numerical computations were performed using 64-bit floating point `Doubles`. In order to deal with !-boxes, a simple post-processing algorithm was introduced (which was not considered notable enough for chapter 3): overlapping vertices (which Quantomatic's rewriting engine generates when matching with !-box `Rules`) are simply detected and moved towards their adjacent vertices (in the direction of the vector mean) by $\frac{1}{3}$ of the distance to the nearest adjacent vertex.

However, the rewrite rule-based drawing algorithm can also be applied to whole `Derivations`. Rather than implement new classes `RuleDerivationLayoutStrategy` and `RuleDerivationLayout` for this case, I implemented general *lifting* classes. In particular, `LiftedDStepLayoutDerivationLayoutStrategy` and `LiftedDStepLayoutDerivationLayout` extend any layout algorithm for single proof steps to a layout algorithm for whole proofs in the same way this is done for rewrite rule-based drawing in §3.4. Furthermore, `LiftedGraphLayoutDStepLayoutStrategy` and `LiftedGraphLayoutRuleLayoutStrategy` extend any `GraphLayoutStrategy` to a `RuleLayoutStrategy` or `DStepLayoutStrategy` respectively, simply by applying the `GraphLayoutStrategy` to both left-hand and right-hand sides. `LiftedDStepLayoutDerivationLayoutStrategy` and `LiftedGraphLayoutDStepLayoutStrategy` can even be composed in order to extend a `GraphLayoutStrategy` to a `DerivationLayoutStrategy`. Ultimately, being able to *lift* layout algorithms from smaller to larger document structures shows the compositionality and functoriality of the document data structures.

[2]paths of edges connected by *wire vertices* and starting and ending at non-wire vertices, including the case of a single edge between non-wire vertices

4.3 Enhanced graphical user interface

I enhanced the graphical user interface for editing `Derivation`s. Architecturally, the existing Quantomatic GUI is mostly inspired by the Model-View-Controller pattern. I therefore made changes in the classes `gui.DerivationPanel` and `gui.DerivationController`, representing respectively View and Controller.

For drawing of whole `Derivation`s, I chose to introduce a new submenu under the 'Derivation' menu, replacing the previously existing menu item which used the legacy `DeriveLayout` code. By maintaining the same location, existing users are less disturbed. Two menu items are currently available, one using the legacy `ForceClusterDerivationLayoutStrategy` and one using my new algorithm, `RuleDStepLayoutStrategy` with `LiftedDStepLayoutDerivationLayoutStrategy`. The menu itself is defined in `QuantoDerive.DeriveMenu.LayoutDerivation`, and calls out to `DerivationController.layoutDerivation`, which has been abstracted to take a `DerivationLayoutStrategy` parameter. I deliberately kept this implementation general, so adding new algorithms (that implement the `DerivationLayoutStrategy` interface) to the menu is easy: all you need is to duplicate a single line and change the name and `DerivationLayoutStrategy` parameter.

I also added two new toolbar buttons, to enable the layout of individual proof steps in a derivation. The first button simply applies `RuleDStepLayoutStrategy` to the currently shown `DStep`, replacing the layout of both the LHS and the RHS. The second applies `RuleDStepLayoutStrategy`, replacing the layout of the RHS, but inserting a new, empty proof step before the current proof step which has no logical rewriting, but merely changes layout from the old LHS layout to the new LHS layout. Both buttons are only available when editing a `DStep`, and are disabled when editing the root or a head of the `Derivation`. To implement these additions, in the View, new attributes `DerivationPanel.LayoutStepButton` and `DerivationPanel.InsertLayoutStepButton` of type `scala.swing.Button` were created, along with appropriate custom-drawn icons, and appended to `DerivationPanel.derivationButtons` and `DerivationPanel.DeriveToolbar`. In the Controller, click handlers were added to `DerivationController.reactions` and logic was added to `DerivationController.state_` so that the buttons would be disabled and enabled as described.

Chapter 5

Experimental Evaluation

5.1 Benchmarking

It's important to experimentally verify that the layout system is efficient and responsive. This shows both that the algorithm and implementation are efficient and that the implementation is usable in practial scenarios.

5.1.1 Method

1. Derivations are identified and selected for this experiment. I intentionally selected for variation, and proof length.

2. The program was started, and a project was loaded, and the whole-derivation rewrite rule-based drawing algorithm was executed 25 times on randomly picked derivations. This is to 'warm up' the Java Just-In-Time compiler and avoid sudden performance spikes as the code is compiled to native code and optimised.

3. The program was timed while executing the rewrite rule-based drawing algorithm on each whole derivation. The timing is performed by patching the `DerivationController.layoutDerivation` method to measure times with nanosecond precision. Each derivation is closed and re-opened twice so that the timing can be performed three times.

4. I analysed the timings.

The proofs selected in step 1 were all taken from Quantomatic sample project data. Note that for variety, roughly half of the derivations selected were for ZX-calculus, which is the most popular graphical calculus in Quantomatic, roughly half were for ZH-calculus, which is the other major (and more theoretically general) graphical calculus in Quantomatic, and one was from Spekkens' toy theory for ZX-calculus.

Sample project	**Derivation**
zx-stabilizer	sample_simplified
zx-stabilizer	rotate_lem
zx-stabilizer	rotate
zx-stabilizer	gss_cc(n,n)
zh	n-disconnect
zh	gen-n-disconnect
zh	disconnect-4
zh	gen-disconnect-4
spekkens	HadamardAnnihilation

5.1.2 Results

Sample project	**Derivation**	**Times (nanoseconds)**		
zx-stabilizer	sample_simplified	10710739	9817536	10516417
zx-stabilizer	rotate_lem	5053901	5051618	5688947
zx-stabilizer	rotate	5213494	2247499	4300676
zx-stabilizer	gss_cc(n,n)	4283259	4184278	5156822
zh	n-disconnect	8380444	8594880	8901109
zh	gen-n-disconnect	7222373	6057077	6821178
zh	disconnect-4	5977731	8137147	7609095
zh	gen-disconnect-4	7032735	8495643	7361482
spekkens	HadamardAnnihilation	6180974	4462645	4807371

5.1.3 Analysis

The rewrite rule-based layout algorithm, including glue code, never took more than 11 milliseconds for any of the derivations listed above, not even the large ones. If the derivation 'zx-stabilizer/sample_simplified' is ignored, then the new algorithm, including glue code, never took more than 9 milliseconds. Both of these values are clearly less than even $\frac{2}{3}$ of the time between consecutive refreshes of a 60 Hz computer display (occurring, approximately, every $16\frac{2}{3}$ milliseconds). Therefore, it is clear that the new layout algorithm performs well, with no delay to the user experience.

5.2 Human quality evaluation

More importantly, we should experimentally verify that the layout produced by the new proof layout system is preferred by people to the layouts produced by the previous Quantomatic layout algorithm.

5.2.1 Method

1. Short equational proofs (up to 3 steps) are selected for this experiment (see listing below). Variety was intentionally sought. The limit of 3 steps is to prevent fatigue among survey participants, and to enable a wider variety of proofs to be displayed.

2. The equations were laid out using two algorithms: the Quantomatic layout algorithm (force-based layout with clustering) prior to this thesis (as the previous best algorithm), and the new layout algorithm (rule-based layout) from this thesis.

3. Each graph in the equations was rendered to an image using Quantomatic's Java2D display renderer, with highlighting from the left hand side of each rule. The zoom level was kept the same throughout each equation laid out, unless one graph in the equation differed, in which case this graph was presented at its own appropriate zoom level. Note that the background grid shows the zoom level.

4. The rendered equations were then laid out on the pages of the survey, using one page for each question. (The names of the equations from the sample projects were not

laid out on the page.) Where possible, each algorithm was given equal space and a consistent scale in each question. The order of the algorithms in each question was randomised. (Pages of the survey could be landscape or portrait.)

5. The survey was distributed to participants (self-selected members of the University[1]), who then completed and returned the survey. Informed consent was requested from participants for the survey (which received ethical approval from the Computer Science Departmental Research Ethics Committee, reference CS_C1A_19_022). The survey did not collect any personal data, and the completed, received surveys are anonymous. Each participant was asked to circle the layout that they found more readable. Participants were not informed of which algorithms produced the layouts.

6. Returned surveys were then aggregated. Each question was aggregated separately, counting the number of votes for each algorithm.

7. I analysed the data.

The proofs selected in step 1 were all taken from Quantomatic sample project data. Note that for variety, half of the proofs selected were for ZX, and the other half were for ZH-calculus. The randomised order for each question, selected in step 4, is also listed.

Q	**Sample project**	**Derivation**	**'First' layout**	**'Second' layout**
1	zx-cliffordt	HadamardAnn	Force-based	Rule-based
2	zx-qutrit-stabilizer	S1-red	Rule-based	Force-based
3	zh	gen-split-example1	Rule-based	Force-based
4	zh	sp-lem1	Force-based	Rule-based

5.2.2 Results

In total, 9 responses were received. Unfortunately, on one of the received responses, the statement of participant consent (on the survey cover sheet) was not circled (or marked in any other way that showed agreement to consent), and therefore this response was discarded. This left 8 valid responses.

[1]Participation was voluntary and without any reward of any kind. Both existing Quantomatic users and non-Quantomatic users were permitted to participate; this is intentional, since existing Quantomatic users might prefer the previous algorithm out of familiarity.

The per-question aggregated results are:

Q	**Sample project**	**Derivation**	**Force-based**	**Rule-based**
1	zx-cliffordt	HadamardAnn	5	3
2	zx-qutrit-stabilizer	S1-red	7	1
3	zh	gen-split-example1	6	2
4	zh	sp-lem1	5	3

5.2.3 Analysis

A statistical hypothesis test is performed for each question on the number of votes N for rewrite rule-based drawing, at the 5% level of significance. (Note that the independent variable is the layout algorithm and the dependent variable is the voting distribution. The derivation to be laid out is thus a control variable, so each question must be analysed separately.) The null hypothesis H_0 is that participants were indifferent about the derivation layout algorithm, i.e. $N \sim \text{Bin}(8, \frac{1}{2})$ $(p = \frac{1}{2})$. The alternative hypothesis H_1 is that participants preferred one of the algorithms, i.e. $p \neq \frac{1}{2}$. This is a two-tailed test.

The critical values for the two-tailed test are 0 and 8, because if $N \sim \text{Bin}(8, \frac{1}{2})$, then:

$$\mathbb{P}(N \leq 0) = \mathbb{P}(N \geq 8) = 0.00390625 \leq 0.025$$

$$\mathbb{P}(N \leq 1) = \mathbb{P}(N \geq 7) = 0.03515625 > 0.025$$

Therefore 0 is the largest value c for which $\mathbb{P}(N \leq c) \leq 0.025 = \frac{0.05}{2}$, and 8 is the smallest value d for which $\mathbb{P}(N \geq d) \leq 0.025 = \frac{0.05}{2}$. (The value of 0.05 comes from the 5% level of significance.)

Applying the critical value to each question, we see that rewrite rule-based drawing always received between 1 and 7 votes inclusive. The number of votes was never 0 or 8 (or below 0 or above 8). Therefore, on each question, the conclusion is that participants were indifferent about the derivation layout algorithm.

Performing a meta-analysis across all questions, with the null hypothesis H_0 that participants were indifferent about the derivation layout algorithm, i.e. the sum of votes for rewrite rule-based drawing $M \sim \text{Bin}(32, \frac{1}{2})$ $(p = \frac{1}{2})$, and the alternative hypothesis H_1

that participants preferred one of the algorithms, i.e. $p \neq \frac{1}{2}$, the critical values for a 5% level of significance are 9 and 23 respectively. In this case, $M = 3 + 1 + 2 + 3 = 9$ and so the meta-analysis finds that (across different derivations) participants did prefer one of the algorithms, specifically, force-based drawing.

Overall, although no algorithm was preferred by participants for individually considered proofs, when participants considered multiple proofs, they slightly preferred force-based drawing.

I acknowledge that there is definitely room for improvement in the rewrite rule-based drawing algorithm, as particularly shown by Question 2, where several vertices were placed close enough to a !-box to trigger warnings from the Quantomatic renderer, seriously compromising what would otherwise be a successful example of the assumptions and behaviours of rewrite rule-based drawing. However, the methodology had weaknesses which particularly impacted rewrite rule-based drawing. My algorithm was designed primarily to handle derivations using large, complex rewrite rules, such as in figure 3.2. However, Questions 1 and 4 only used rules where every vertex was in a straight line. As the straight line is a simple shape and therefore difficult to identify within a larger graph, rewrite rule-based drawing generally performs less well with it. In Question 4, it produced a good output which emphasised local symmetry, comparable to the force-based layout algorithm, which emphasised global symmetry, but in Question 1, rewrite rule-based drawing was noticeably less successful. Furthermore, almost all of the rules involved in the derivations used in the questions had at most 2 non-boundary vertices on the left-hand side. This triggers the non-general affine transformation cases, which intrinsically performs less well since it does not have enough information to determine the correct layout of the rewritten subgraphs, and can only use weak scale-and-translate or even translate-only approximations.

Chapter 6

Conclusion

In this thesis, I have developed a new layout system for understandable proof layout for the Quantomatic theorem prover. This has included defining the problem, its scope, and its specific goals, designing novel, semantically meaningful algorithms, with analysis of how the specific goals are fulfilled and of time and space complexity, and efficiently implementing the most promising new algorithm, rewrite rule-based drawing, within a tidy, effective architecture for all layout algorithms in Quantomatic, enabling future developments in the field of graphical proof layout.

Rewrite rule-based drawing can be adapted for use in another major field: chemistry. Molecules (and similar entities, e.g. complexes, molecular ions, molecular radicals) can be drawn using *structural formulae*: diagrams on the page, with (typically) points representing atoms and lines representing covalent bonds. *Reaction equations* describe processes that transform *reactant* molecules into *product* molecules. In organic chemistry, molecules are often large and structurally complex, with recognisable, frequently occurring substructures, such as benzene (and similar) rings, the carboxylic acid group, and the phosphate group. Moreover, synthesis of complex organic molecules often involves multiple reactions, using many intermediate products, reactants, and catalysts. On the page, these reactions are often written in series, with no repetition of intermediate products: each intermediate product immediately becomes the reactant for the next reaction. However, each reaction often changes only one or a few *functional groups* on the molecule, building the molecule from its pieces. The strong similarities between these properties of equations for organic synthesis processes and the characteristics identified in §3.2 (bar symmetry) show that rewrite rule-based drawing is usable in computer-based tools that assist organic chemists.

Furthermore, the core of rewrite rule-based drawing, finding the best affine transformation relating a set of vector pairs, has immediate applications in machine learning: supervised learning of a function $\mathbb{R}^m \to \mathbb{R}^n$ using a set $\subseteq \mathbb{R}^m \times \mathbb{R}^n$ of training data, where the function is modelled as an affine transformation and the metric for $\mathbb{R}^n$ is taken to be Euclidean. Using the same techniques as in §3.4, the affine transformation can be computed by solving n systems of $m+1$ linear equations, with $O(m^{2.373\ldots}n)$ time complexity, which is a significant improvement on the naïve technique of solving one system of $n(m+1)$ linear equations, with $O(m^{2.373\ldots}n^{2.373\ldots})$ time complexity.

Rewrite rule-based drawing was most effective under the assumptions (and thus in the situations) which it was designed for. Specifically, the algorithm performed well with long proofs containing rewrites using large rules with clear well-known layouts, such as `zh`'s `disconnect`. The algorithm performed less well on proof steps where the rule's left-hand side contained less than 3 vertices and all vertices were arranged in a straight line, such as `zh`'s `z-id`. In these cases, the general affine transformation could not be identified sensibly, leading the algorithm to use arbitrary restrictions of the classes of affine transformation, producing a suboptimal result in many (but not all) cases. However, the algorithm definitely met the most important criterion that was identified: the stasis of non-rewritten parts. The algorithm also performed well at the other criteria of common substructures and symmetry. Nonetheless, modifications to improve performance regarding these criteria might be possible. However, these criteria that were identified in §3.2 and that the algorithm was then designed to address did not capture all aspects of a desirable layout. Other important criteria include syntactic concerns of graph layout, such as unambiguity (achieved by separating vertices, increasing angles between edges, etc.) and higher-level semantic concepts in the graphical calculi, such as taking advantage of the amalgamating and de-amalgamating nature of spider nodes in order to establish visual grouping (which is particularly important when using rewrite rules that introduce or remove spider nodes).

Hosting the implementation inside Quantomatic had benefits, but also created challenges. One benefit is that Quantomatic is used regularly by researchers working with graphical calculi, and this project has immediately improved their experience by revamping internal code and offering rewrite rule-based derivation drawing. Another benefit is that foundational code, such as graph and derivation data structures and the double-pushout rewriting algorithm, did not need to be re-implemented. The Scala programming language, in which Quantomatic is implemented, also shares many similarities with Java, such as a strong object-oriented type system, which enabled many errors to be discovered at compile-time, rather than at run-time, which would require more effort to uncover. Java-style object-oriented functionality, such as abstract classes and concrete implementations, was also crucial to implementing the new, clean, extensible layout architecture. However, reusing Quantomatic's framework was also problematic because the existing codebase, even inside the trusted computing base, is poorly documented: comments are uncommon and many variable names are ambiguous. For example, the `Graph` class, one of the most foundational, only make any attempt at documenting the interface to 33 of its 101 public methods, and only 12 were documented systematically, describing parameters and return values. Furthermore, Quantomatic's use of the SBT compilation tool for Scala obstructed the development of the implementation by taking roughly 3 minutes to recompile the program each time.[1]

Although it was not the subject of an extended discussion in this thesis, the software implementation was indeed tested at multiple stages during its development, using both automated and manual techniques, in order to verify the correct functioning and suitability of the system. However, this work was felt to be too trivial and implicit in the provision of an implementation which was evaluated in more sophisticated ways, including performance benchmarking and opinion surveys with human participants. As such, these were the topics that chapter 5 focused on.

The evaluation of rewrite rule-based drawing occurred both on a theoretical level, where properties of the algorithm, such as its time and space complexity, and criteria fulfilment, were examined during the design phase, and on a practical level, where the implementation's performance was benchmarked and human participants were asked to compare the output of rewrite rule-based drawing with the output of another algorithm for the task. Typical

[1]This is caused by Scala's design impeding effective incremental recompilation. Unlike in Java, compiled class files and source files do not correspond. Consequently, determining which source files to recompile cannot be done by comparing modification times between compiled class files and source code files.

research in programming language design lacks rigorous empirical evidence for effectiveness [Stefik and Hanenberg, 2017], and this also holds of the closely related field of interactive theorem proving. In particular, independent human participants rarely take part in interactive theorem proving research. On the other hand, I performed a study, with approval from the Research Ethics Committee, into whether people prefer rewrite rule-based drawing to the prior best available algorithm, force-based rewriting with clusters. This provided robust scientific evidence, and also a high ethical standard of scientific conduct. Further rigour was achieved by using statistical hypothesis testing to determine whether a significant result had occurred, instead of simplistic claims about 'how many people liked it'.[2] Therefore, this thesis, by presenting both theoretical and experimental evidence for the effectiveness of its proposed algorithm, exceeds the typical level of scientific rigour in the field.

My algorithms are intentionally not derived from existing research in graph drawing, because existing solutions in dynamic graph drawing typically only handle small changes in graphs, and are not intended. Additionally, avoiding existing graph drawing research highlights the design of the algorithm around semantically meaningful notions, such as compositionality, as opposed to syntactic notions, which prevail in existing graph drawing research. Nonetheless, a hybrid approach could provide the benefits of both syntactic and semantic paradigms, optimising for the goals of both, including stasis of non-rewritten parts, similarity of substructures, symmetry, equal spacing, similar line length, equally spaced angles around each vertex, etc. Such a hybrid could produce output that users prefer to both rewrite rule-based drawing and existing syntactic graph drawing algorithms (which performed similarly in §5.2), and thus outperform both.

In wider society, many symbols have non-mathematical meanings. Some symbols have offensive or discriminatory meanings. In order to be useful to the widest possible audience, new layout algorithms, or adaptations of existing ones, could be designed to prevent inappropriate graph layouts. Such an algorithm might avoid solutions containing inappropriate well-known layouts, contrasting with rewrite rule-based drawing, which aimed to incorporate (appropriate) well-known layouts.

[2]More worrying are claims about 'how many students liked the new technology when they were taking a course where they would get a higher grade for using the new technology'.

Appendix A

Layout interface specification

```
/// Builds a GraphLayoutData object containing the computed layout for a
    graph.
abstract class GraphLayoutStrategy {
  /// Secondary constructor, which allows the Graph to be provided at
      construction.
  def this(graph : Graph) {
    this();
    this.setGraph(graph)
  }

  /// Provide a Graph for which the layout can be computed with
      DStepLayoutStrategy.layout().
  /// @param graph the Graph that is to be laid out
  def setGraph(graph : Graph) : Unit;

  /// Implementations may provide additional methods for setting parameters
       for the layout algorithm.
  /// Implementations may also provide additional constructors for setting
      parameters for the layout algorithm.

  /// Create a GraphLayoutData object containing the layout data for the
      graph.
  /// The graph must have been previously provided to the
      GraphLayoutStrategy (by constructing with the data or by calling
      GraphLayoutStrategy.setGraph).
  /// @return a GraphLayoutData containing the layout data for the graph.
  def layout() : GraphLayoutData;
}

/// Represents the layout of a Graph.
/// Users do not construct objects of this class directly, but obtain
    instances from a GraphLayoutStrategy instead.
```

```
/// Note that for legacy reasons, although the GraphLayout class represents
     the Strategy design pattern, it does not generate GraphLayoutData
    objects.
abstract class GraphLayoutData {
  /// Get the layout's coordinates for the vertex named v.
  /// @param v the name of the vertex in the graph selected by dso
  /// @return the layout's coordinates for the queried vertex.
  def getCoords(v: VName) : (Double, Double);

  /// The original Graph, with the previous layout, that this layout was
      computed for.
  def graph : Graph;

  /// Get a new Graph exactly equal to the input, but having the layout
      from this GraphLayoutData object. The Graph that was computed on is
      not modified. This method is part of the trusted computing base.
  /// @return a new Graph exactly equal to the input, but having the layout
       from this GraphLayoutData object.
  final def asGraph() : Graph = {
    var newGraph = graph
    // For each vertex in the graph, replace its coordinate with the new
       coordinate of the vertex.
    for (v <- graph.verts) {
      newGraph = newGraph.updateVData(v) { vd => vd.withCoord(getCoords(v))
          }
    }
    return newGraph
  }
}

/// Builds GraphLayoutData objects containing the computed layouts for a
    rule.
abstract class RuleLayoutStrategy {
  /// Secondary constructor, which allows the Rule to be provided at
      construction.
  def this(rule : Rule) {
    this();
    this.setRule(rule)
  }

  /// Provide a Rule for which the layout can be computed with
      RuleLayoutStrategy.layout().
  /// @param rule the Rule that is to be laid out
  def setRule(rule: Rule) : Unit;

  /// Implementations may provide additional methods for setting parameters
       for the layout algorithm.
  /// Implementations may also provide additional constructors for setting
```

```
    parameters for the layout algorithm.

  /// Create a GraphLayoutData object containing the layout data for the
      Rule's RHS graph.
  /// The Rule must have been previously provided to the RuleLayoutStrategy
       (by constructing with the rule or by calling RuleLayoutStrategy.
      setRule).
  /// @return a GraphLayoutData containing the layout data for the output
      of the Rule.
  def layoutRHS() : GraphLayoutData;

  /// Create a GraphLayoutData object containing the layout data for the
      Rule's LHS graph.
  /// The Rule must have been previously provided to the RuleLayoutStrategy
       (by constructing with the rule or by calling RuleLayoutStrategy.
      setRule).
  /// @return a GraphLayoutData containing the layout data for the source
      graph.
  def layoutLHS() : GraphLayoutData;
}

/// Builds GraphLayoutData objects containing the computed layouts for a
    derivation step.
abstract class DStepLayoutStrategy {
  /// Secondary constructor, which allows the DStep to be provided at
      construction.
  def this(step : DStep, sourceGraph : Graph) {
    this();
    this.setStep(step, sourceGraph)
  }

  /// Provide a DStep for which the layout can be computed with
      DStepLayoutStrategy.layout().
  /// @param step the DStep that is to be laid out
  /// @param sourceGraph the source graph of the derivation step
  def setStep(step : DStep, sourceGraph : Graph) : Unit;

  /// Implementations may provide additional methods for setting parameters
       for the layout algorithm.
  /// Implementations may also provide additional constructors for setting
      parameters for the layout algorithm.

  /// Create a GraphLayoutData object containing the layout data for the
      output of the DStep.
  /// The DStep and source graph must have been previously provided to the
      DStepLayoutStrategy (by constructing with the data or by calling
      DStepLayoutStrategy.setStep).
  /// @return a GraphLayoutData containing the layout data for the output
```

```
      of the DStep.
  def layoutOutput() : GraphLayoutData;

  /// Create a GraphLayoutData object containing the layout data for the
      source graph.
  /// The DStep and source graph must have been previously provided to the
      DStepLayoutStrategy (by constructing with the data or by calling
      DStepLayoutStrategy.setStep).
  /// @return a GraphLayoutData containing the layout data for the source
      graph.
  def layoutSource() : GraphLayoutData;
}

/// Builds a DerivationLayout object containing the computed layout for a
    derivation.
abstract class DerivationLayoutStrategy {
  /// Secondary constructor, which allows the Derivation to be provided at
      construction.
  def this(derivation : Derivation) {
    this();
    this.setDerivation(derivation)
  }

  /// Provide a Derivation for which the layout can be computed with
      DerivationLayoutStrategy.layout().
  /// @param derivation the Derivation that is to be laid out
  def setDerivation(derivation : Derivation) : Unit;

  /// Implementations may provide additional methods for setting parameters
       for the layout algorithm.
  /// Implementations may also provide additional constructors for setting
      parameters for the layout algorithm.

  /// Create an object of type DerivationLayout containing the data of the
      new derivation layout.
  /// A Derivation must have been previously provided to the
      DerivationLayoutStrategy (by constructing with a Derivation or by
      calling DerivationLayoutStrategy.setLayout).
  /// @return a DerivationLayout corresponding to the previously provided
      Derivation
  def layout() : DerivationLayout;
}

/// Represents the layout of all the graphs in a Derivation.
/// Users do not construct objects of this class directly, but use the
    DerivationLayoutStrategy instead.
abstract class DerivationLayout {
  /// Get the layout's coordinates for the vertex named v in: the root
```

```
    graph if dso is None, or the graph generated by derivation step ds if
    dso is Some(ds)
  /// @param dso None to query the root graph, otherwise the DSName of the
    DStep generating the graph to query
  /// @param v the name of the vertex in the graph selected by dso
  /// @return the layout's coordinates for the queried vertex.
  def getCoords(dso : Option[DSName], v: VName) : (Double, Double);

  /// The Derivation that this layout was computed for.
  def derivation : Derivation;

  /// Get a new Derivation exactly equal to the input, but having the
    layout from this DerivationLayout object. The Derivation that was
    computed on is not modified. This method is part of the trusted
    computing base.
  /// @return a new Derivation exactly equal to the input, but having the
    layout from this DerivationLayout object.
  final def asDerivation() : Derivation = {
    def asGraph(dso : Option[DSName], graph : Graph) : Graph = {
      var newGraph = graph
      // For each vertex in the graph, replace its coordinate with the new
        coordinate of the vertex.
      for (v <- graph.verts) {
        newGraph = newGraph.updateVData(v) { vd => vd.withCoord(getCoords(
          dso, v)) }
      }
      return newGraph
    }
    // Insert the new layout of the root graph.
    var newDerivation = derivation.copy(root = asGraph(None, derivation.root
      ))
    // For each derivation step, insert the new layout of the graph
      generated by that derivation step.
    for (ds <- newDerivation.steps.keys) {
      newDerivation = newDerivation.updateGraphInStep(ds, asGraph(Some(ds),
        derivation.steps(ds).graph))
    }
    return newDerivation
  }
}
```

Appendix B

Layout implementation

All interfaces and implementations, both for layout and for the GUI, are described in chapter 4. To prevent this thesis from becoming too long, only `RuleDStepLayoutStrategy`, `RuleDStepLayout`, `LiftedDStepLayoutDerivationLayoutStrategy` and `LiftedDStepLayoutDerivationLayout` have been presented below.

```
/// Implements DStepLayoutStrategy which uses the layouts from each Rule to
    compute the layout of a Derivation.
class RuleDStepLayoutStrategy extends DStepLayoutStrategy {
  var step : DStep = null
  var root : Graph = null

  var automaticVertexSeparationEnabled : Boolean = true

  // Cache of the optimal affine transformation for the pair (step, root)
     above.
  var affineTransform : Option[((Double, Double)) => (Double, Double)] =
     null

  /// Provide a DStep for which the layout can be computed with
     DStepLayoutStrategy.layout().
  /// @param step the DStep that is to be laid out
  /// @param sourceGraph the source graph of the derivation step
  def setStep(step : DStep, root : Graph) : Unit = {
    this.step = step
    this.root = root
    this.affineTransform = null // Clear the cached optimal affine
       transformation.
  }
```

```
/// Set whether to automatically separate vertices in the output with
    equal coordinates. When the matching algorithm matches a subgraph of
    the source graph with a rule containing !-boxes, the vertices
    generated by expanding the !-box are all placed with equal coordinates
    . This layout is then copied into the target diagram.
/// @param enabled true to enable this behaviour
def setAutomaticVertexSeparation(enabled : Boolean) : Unit = {
  automaticVertexSeparationEnabled = enabled
}

/// Compute affineTransform for the present values of (step, root).
private[this] def computeAffineTransform() : Unit = {
  this.affineTransform = null
  // Compare the layout of the LHS of the rule applied with the layout of
      the match in the source (matched) graph. Compute the optimal affine
      transformation that minimises the sum of squared distance between
      each vertex of the LHS of the rule and its match in the source graph.
       The thesis contains a derivation of the below method.
  val verts = (step.rule.lhs.verts intersect root.verts).toArray // The
      vertices need to be in a consistent order for the following
      numerical computations, which read coordinates elementwise.
  if (verts.size == 0) {
    // If there are no vertices to transform, don't bother.
    this.affineTransform = None
  }
  else if (verts.size == 1) {
    // Translate only.
    // Don't bother with least-squares optimisation here. Just pick the
        first vertex.
    val v = verts.head
    val (x1, y1) = root.vdata(v).coord
    val (x0, y0) = step.rule.lhs.vdata(v).coord
    this.affineTransform = Some(RuleDStepLayoutStrategy.affineTransform
        (1.0, 0.0, 0.0, 1.0, x1 - x0, y1 - y0))
  }
  else if (verts.size >= 2) {
    // The optimisation is expressed in terms of linear algebra. See the
        thesis for a derivation.
    val nverts : Double = verts.size.toDouble
    val Wx : linalg.DenseVector[Double] = linalg.DenseVector(verts.map(v =
        > step.rule.lhs.vdata(v).coord._1))
    val Wy : linalg.DenseVector[Double] = linalg.DenseVector(verts.map(v =
        > step.rule.lhs.vdata(v).coord._2))
    val Zx : linalg.DenseVector[Double] = linalg.DenseVector(verts.map(v =
        > root.vdata(v).coord._1))
    val Zy : linalg.DenseVector[Double] = linalg.DenseVector(verts.map(v =
        > root.vdata(v).coord._2))
    val sumWx : Double = linalg.sum(Wx)
val sumWy : Double = linalg.sum(Wy)
val sumZx : Double = linalg.sum(Zx)
val sumZy : Double = linalg.sum(Zy)
val sumWxWx : Double = Wx.dot(Wx)
val sumWyWy : Double = Wy.dot(Wy)
val sumWxZx : Double = Wx.dot(Zx)
val sumWxZy : Double = Wx.dot(Zy)
val sumWyZx : Double = Wy.dot(Zx)
val sumWyZy : Double = Wy.dot(Zy)
if (verts.size >= 3) {
  // Fully generic affine transformation.
  val sumWxWy : Double = Wx.dot(Wy)
  val matrix : linalg.DenseMatrix[Double] = linalg.DenseMatrix(List(
     sumWxWx, sumWxWy, sumWx),
                                                          List(sumWxWy,
                                                             sumWyWy,
                                                             sumWy),
                                                          List(sumWx,
                                                             sumWy,
                                                             nverts))
  val matchVector1 : linalg.DenseVector[Double] = linalg.DenseVector[
     Double](sumWxZx, sumWyZx, sumZx)
  val matchVector2 : linalg.DenseVector[Double] = linalg.DenseVector[
     Double](sumWxZy, sumWyZy, sumZy)
  try {
    val output1 : linalg.DenseVector[Double] = matrix \ matchVector1 //
        a, b, e
    val output2 : linalg.DenseVector[Double] = matrix \ matchVector2 //
        c, d, f
    this.affineTransform = Some(RuleDStepLayoutStrategy.
       affineTransform(output1(0), output1(1), output2(0), output2(1),
        output1(2), output2(2)))
  } catch {
    case _ : linalg.MatrixSingularException => {}
  }
}
if (this.affineTransform == null) {
  // Scale, rotate, and translate only (i.e. [[a, b], [-b, a]] @ w + [
     e, f]).
  val matrix : linalg.DenseMatrix[Double] = linalg.DenseMatrix(List(
     sumWxWx + sumWyWy, 0.0, sumWx, sumWy),
                                                          List(0.0,
                                                             sumWxWx +
                                                             sumWyWy,
                                                             sumWy, -
                                                             sumWx),
                                                          List(sumWx,
                                                             sumWy,
                                                              nverts, 0.0),

                                                           List(sumWy, -
                                                              sumWx, 0.0,
                                                              nverts))
      val vector : linalg.DenseVector[Double] = linalg.DenseVector[Double
         ](sumWxZx + sumWyZy, sumWyZx - sumWxZy, sumZx, sumZy)
      try {
        val output : linalg.DenseVector[Double] = matrix \ vector // a, b,
             e, f
        this.affineTransform = Some(RuleDStepLayoutStrategy.
           affineTransform(output(0), output(1), -output(1), output(0),
           output(2), output(3)))
      } catch {
        case _ : linalg.MatrixSingularException => {}
      }
    }
    if (this.affineTransform == null) {
      // Scale and translate only (i.e. [[a, 0], [0, a]] @ w + [e, f]).
      val matrix : linalg.DenseMatrix[Double] = linalg.DenseMatrix(List(
         sumWxWx + sumWyWy, sumWx, sumWy),
                                                           List(sumWx,
                                                              nverts, 0.0),

                                                           List(sumWy, 0.0,
                                                               nverts))
      val vector : linalg.DenseVector[Double] = linalg.DenseVector[Double
         ](sumWxZx + sumWyZy, sumZx, sumZy)
      try {
        val output : linalg.DenseVector[Double] = matrix \ vector // a, e,
             f
        this.affineTransform = Some(RuleDStepLayoutStrategy.
           affineTransform(output(0), 0.0, 0.0, output(0), output(1),
           output(2)))
      } catch {
        case _ : linalg.MatrixSingularException => {}
      }
    }
    if (this.affineTransform == null) {
      // Translate only (i.e. [[1, 0], [0, 1]] @ w + [e, f]).
      // This least-squares minimisation has a simple form: the mean.
      // This is guaranteed to successfuly produce an affine
         transformation.
      this.affineTransform = Some(RuleDStepLayoutStrategy.affineTransform
         (1.0, 0.0, 0.0, 1.0, sumZx / nverts, sumZy / nverts))
    }
  }
}
```

```
/// Post-process a set of coordinates and the vertices to separate
   vertices with the same coordinates.
private[this] def separateVerticesAutomatically(graph : Graph, coords :
   mutable.HashMap[VName, (Double, Double)]) : mutable.HashMap[VName, (
   Double, Double)] = {
  // Invert the coords map to find vertices with the same coordinates.
  val vertsAt = new mutable.HashMap[(Double, Double), mutable.HashSet[
     VName]]()
  for ((vert, coord) <- coords) {
    if (!vertsAt.contains(coord)) {
      vertsAt(coord) = new mutable.HashSet[VName]()
    }
    vertsAt(coord) += vert
  }
  // Separate the vertices.
  val layout = new RuleGraphLayoutData(graph, coords)
  val newCoords = new mutable.HashMap[VName, (Double, Double)]()
  newCoords ++= coords
  for ((coord, verts) <- vertsAt) {
    // Skip vertices which have unique coordinates.
    if (verts.size > 1) {
      // Calculate the adjacent vertices of each vertex (which do not
         share the coordinate), and the least distance to each such
         adjacent vertex.
      for (vert <- verts) {
        // Calculate the adjacent vertices of each vertex (which do not
           share the coordinate).
        val adjacents = graph.adjacentVerts(vert) &~ verts
        // Calculate the least distances and cumulative distances to each
           adjacent vertex.
        var leastSqrDistance : Option[Double] = None
        var sdx = 0.0
        var sdy = 0.0
        for (adjVert <- adjacents) {
          val dx = coord._1 - layout.getCoords(adjVert)._1
          val dy = coord._2 - layout.getCoords(adjVert)._2
          sdx += dx
          sdy += dy
          val sqrDistance = dx * dx + dy * dy
          if (leastSqrDistance.isEmpty || sqrDistance < leastSqrDistance.
             get) {
            leastSqrDistance = Some(sqrDistance)
          }
        }
        if (!leastSqrDistance.isEmpty) {
          // Move each vertex towards its adjacent vertices by at most 1/3
             of the minimum distance to each such adjacent vertex.
          val moveDistance = scala.math.sqrt(leastSqrDistance.get) / 3.0
          val scaleFactor = moveDistance / scala.math.sqrt(sdx * sdx + sdy
              * sdy)
          val changeLimit = moveDistance * 60.0
          newCoords(vert) = (coord._1 + (((sdx * scaleFactor) min
              changeLimit) max (-changeLimit)), coord._2 + (((sdy *
              scaleFactor) min changeLimit) max (-changeLimit)))
        }
      }
    }
  }
  return newCoords
}

/// Create a RuleGraphLayoutData object containing the layout data for
    the output of the DStep.
/// The DStep and source graph must have been previously provided to the
    RuleDStepLayoutStrategy (by constructing with the data or by calling
    RuleDStepLayoutStrategy.setStep).
/// @return a RuleGraphLayoutData containing the layout data for the
    output of the DStep.
def layoutOutput() : GraphLayoutData = {
  // Compute the affine transformation, if not already computed.
  if (affineTransform == null) {
    computeAffineTransform()
  }
  // Apply the affine transformation to the inclusion of the rule's RHS
      inside the target (generated) graph.
  var newCoords = new mutable.HashMap[VName, (Double, Double)]()
  if (!affineTransform.isEmpty) {
    val affineTransform = this.affineTransform.get
    for (v <- (step.rule.rhs.verts intersect step.graph.verts)) {
      newCoords(v) = affineTransform(step.rule.rhs.vdata(v).coord)
    }
  }
  // Apply automatic vertex separation, if enabled.
  if (this.automaticVertexSeparationEnabled) {
    newCoords = separateVerticesAutomatically(step.graph, newCoords)
  }
  return new RuleGraphLayoutData(step.graph, newCoords)
}

/// Create a RuleGraphLayoutData object containing the layout data for
    the source graph.
/// The DStep and source graph must have been previously provided to the
    RuleDStepLayoutStrategy (by constructing with the data or by calling
    RuleDStepLayoutStrategy.setStep).
  /// @return a RuleGraphLayoutData containing the layout data for the
      source graph.
  def layoutSource() : GraphLayoutData = {
    // Compute the affine transformation, if not already computed.
    if (affineTransform == null) {
      computeAffineTransform()
    }
    // Apply the affine transformation to the inclusion of the rule's LHS
       inside the source (matched) graph.
    var newCoords = new mutable.HashMap[VName, (Double, Double)]()
    if (!affineTransform.isEmpty) {
      val affineTransform = this.affineTransform.get
      for (v <- (step.rule.lhs.verts intersect root.verts)) {
        newCoords(v) = affineTransform(step.rule.lhs.vdata(v).coord)
      }
    }
    // Apply automatic vertex separation, if enabled.
    if (this.automaticVertexSeparationEnabled) {
      newCoords = separateVerticesAutomatically(root, newCoords)
    }
    return new RuleGraphLayoutData(root, newCoords)
  }
}

/// Implements a RuleGraphLayout which uses the layouts from each Rule to
    compute the layout of a Graph.
class RuleGraphLayoutData private[this] extends GraphLayoutData {
  private[this] var _graph : Graph = null;
  private[this] var coords : mutable.Map[VName, (Double, Double)] = null;

  /// Not part of the public interface. Constructs a RuleDerivationLayout.
  private[layout] def this(graph : Graph, coords : mutable.Map[VName, (
      Double, Double)]) = {
    this()
    this._graph = graph
    this.coords = coords
  }

  /// Get the layout's coordinates for the vertex named v.
  /// @param v the name of the vertex in the graph selected by dso
  /// @return the layout's coordinates for the queried vertex.
  def getCoords(v: VName) : (Double, Double) = {
    return coords.getOrElse(v, _graph.vdata(v).coord) // Unset coordinates
       remain as-is.
  }

  /// The original Graph, with the previous layout, that this layout was
      computed for.
  def graph : Graph = {
    return _graph
  }
}

object RuleDStepLayoutStrategy {
  /// Compute the 2D affine transformation:
  /// [a b] [e]
  /// [c d] @ coord + [f]
  /// @param a, b, c, d, e, f affine transformation entries as above
  /// @return pair of values equal to above equation
  def affineTransform(a : Double, b : Double, c : Double, d : Double, e :
     Double, f : Double)(coord : (Double, Double)) : (Double, Double) = {
    val x = coord._1
    val y = coord._2
    return (a * x + b * y + e, c * x + d * y + f)
  }
}


/// Builds a LiftedDStepLayoutDerivationLayout object containing the
    computed layout for a derivation.
/// Lifts a DStepLayoutStrategy to a DerivationLayoutStrategy.
class LiftedDStepLayoutDerivationLayoutStrategy extends
    DerivationLayoutStrategy {
  private[this] var strategy : DStepLayoutStrategy = null;
  private[this] var derivation : Derivation = null;

  /// Secondary constructor, which allows the DStepLayoutStrategy to be
     provided at construction.
  def this(strategy : DStepLayoutStrategy) {
    this();
    this.setStrategy(strategy)
  }

  /// Secondary constructor, which allows the DStepLayoutStrategy and
     Derivation to be provided at construction.
  def this(strategy : DStepLayoutStrategy, derivation : Derivation) {
    //this(derivation); // Secondary constructors cannot call superclass
       constructors
    this();
    this.setDerivation(derivation);
    this.setStrategy(strategy)
  }

  /// Provide a DStepLayoutStrategy to lift when computing
     LiftedDStepLayoutDerivationLayoutStrategy.layout().
  /// @param strategy the DStepLayoutStrategy to lift
```

```
def setStrategy(strategy : DStepLayoutStrategy) : Unit = {
  this.strategy = strategy;
}

/// Provide a Derivation for which the layout can be computed with
    LiftedDStepLayoutDerivationLayoutStrategy.layout().
/// @param derivation the Derivation that is to be laid out
def setDerivation(derivation : Derivation) : Unit = {
  this.derivation = derivation;
}

/// Create an object of type LiftedDStepLayoutDerivationLayout containing
     the data of the new derivation layout.
/// A Derivation must have been previously provided to the
    LiftedDStepLayoutDerivationLayoutStrategy (by constructing with a
    Derivation or by calling LiftedDStepLayoutDerivationLayoutStrategy.
    setLayout).
/// @return a LiftedDStepLayoutDerivationLayout corresponding to the
    previously provided Derivation
def layout() : DerivationLayout = {
  var layouts = new mutable.HashMap[Option[DSName], GraphLayoutData]();
  // Lay out each step, in breadth-first order.
  for (ds <- breadthFirstDSteps()) {
    val parentDSO = derivation.parentMap.get(ds);
    // Special case for first iteration, when no graph layout exists for
       the root of the derivation.
    if (parentDSO.isEmpty && !layouts.contains(parentDSO)) {
      strategy.setStep(derivation.steps(ds), derivation.root);
      layouts(Some(ds)) = strategy.layoutOutput();
      layouts(None) = strategy.layoutSource();
    }
    else {
      strategy.setStep(derivation.steps(ds), layouts(parentDSO).asGraph())
          ;
      layouts(Some(ds)) = strategy.layoutOutput();
    }
  }
  return new LiftedDStepLayoutDerivationLayout(derivation, layouts);
}

/// Get a list of the DStep names of a Derivation in breadth-first order,
     excluding the root node.
/// @param derivation
/// @return the DSNames in breadth-first order
private[this] def breadthFirstDSteps() : mutable.Queue[DSName] = {
  val out = new mutable.Queue[DSName]()
  val q = new mutable.Queue[DSName]()
  q ++= derivation.firstSteps
```

```
    while (!q.isEmpty) {
      val ds = q.dequeue()
      out += ds
      q ++= derivation.children(ds) // This assumes that the children are
          unordered and that Derivation objects are trees (hence do not
          contain cycles and therefore will never revisit an already-seen
          vertex).
    }
    return out
  }
}

/// Represents the layout of all the graphs in a Derivation.
/// Users do not construct objects of this class directly, but use the
    LiftedDStepLayoutDerivationLayout instead.
class LiftedDStepLayoutDerivationLayout private extends DerivationLayout {
  private[this] var _derivation : Derivation = null
  private[this] var layouts : mutable.Map[Option[DSName], GraphLayoutData]
      = null

  /// Not part of the public interface. Constructs a
      LiftedDStepLayoutDerivationLayout.
  private[layout] def this(derivation : Derivation, layouts : mutable.Map[
      Option[DSName], GraphLayoutData]) = {
    this()
    this._derivation = derivation
    this.layouts = layouts
  }

  /// Get the layout's coordinates for the vertex named v in: the root
      graph if dso is None, or the graph generated by derivation step ds if
      dso is Some(ds)
  /// @param dso None to query the root graph, otherwise the DSName of the
      DStep generating the graph to query
  /// @param v the name of the vertex in the graph selected by dso
  /// @return the layout's coordinates for the queried vertex.
  def getCoords(dso : Option[DSName], v: VName) : (Double, Double) = {
    return layouts(dso).getCoords(v)
  }

  /// The Derivation that this layout was computed for.
  def derivation : Derivation = {
    return _derivation
  }
}
```

Appendix C

References

Bundy, Alan (editor), Atiyah, Michael (editor), Macintyre, Angus (editor), and MacKenzie, Donald (editor). (2005) 'The nature of mathematical proof: Papers of a Discussion Meeting Issue organized and edited by Alan Bundy, Michael Atiyah, Angus Macintyre and Donald MacKenzie' (special issue), *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 363, no. 1835, pp. 2329-2461. London, United Kingdom: Royal Society.

Bibliography

[Asperti et al., 2007] Asperti, Andrea, Coen, Claudio Sacerdoti, Tassi, Enrico, and Zacchiroli, Stefano. (2007) 'User Interaction with the Matita Proof Assistant', *Journal of Automated Reasoning* vol. 39, no. 2, pp. 109–139. Berlin, Germany: Springer.

[Backens, 2016] Backens, Miriam K. (2016) *Completeness and the* ZX*-calculus.* Doctor of Philosophy Thesis, University of Oxford.

[Backens and Kissinger, 2018] Backens, Miriam and Kissinger, Aleks. (2018) 'ZH: A Complete Graphical Calculus for Quantum Computations Involving Classical Non-linearity', *Proceedings of the 15th Annual Conference on Quantum Physics and Logic (QPL).* Waterloo, New South Wales: Electronic Proceedings in Theoretical Computer Science.

[Branke, 2001] Branke, Jürgen. (2001) 'Dynamic Graph Drawing'. In: Kaufmann, Michael and Wagner, Dorothea (eds.). *Lecture Notes in Computer Science 2025: Drawing Graphs: Methods and Models.* Berlin, Germany: Springer-Verlag. ch. 9.

[Bundy, 2011] Bundy, Alan. (2011) 'Automated theorem provers: a practical tool for the working mathematician?', *Annals of Mathematics and Artificial Intelligence* vol. 61 no. 1 pp. 3–14. Berlin, Germany: Springer.

[Coecke and Kissinger, 2017] Coecke, Bob and Kissinger, Aleks. (2017) *Picturing Quantum Processes.* Cambridge, England: Cambridge University Press.

[Cohen, 1963] Cohen, Paul J. (1963) 'The Independence of the Continuum Hypothesis', *Proceedings of the National Academy of Sciences of the United States of America* vol. 50, no. 6, pp. 1143–1148. Washington, D.C.: National Academy of Sciences.

[Davie and Stothers, 2013] Davie, Alexander M. and Stothers, Andrew James. (2013) 'Improved bound for complexity of matrix multiplication', *Proceedings of the Royal Society of Edinburgh: Section A: Mathematics*, vol. 143 no. 2 pp. 351–369. Cambridge, England: Cambridge University Press.

[DeMillo, Lipton, and Perlis, 1979] DeMillo, Richard A., Lipton, Richard J., and Perlis, Alan J. (May 1979) 'Social processes and proofs of theorems and programs', *Communications of the ACM*, vol. 22, no. 5, pp. 271–280. Cited in Pierce, Benjamin C. (2002) *Types and Programming Languages.* Cambridge, Massachusetts: MIT Press.

[Derksen and Weyman, 2005] Derksen, Harm and Weyman, Jerzy. (2005) 'Quiver Representations', *Notices of the American Mathematical Society* vol. 52 no. 2 pp. 200–206. Providence, Rhode Island: American Mathematical Society.

[Dixon and Duncan, 2009] Dixon, Lucas and Duncan, Ross. (2009) 'Graphical Reasoning in Compact Closed Categories for Quantum Computation', *Annals of Mathematics and Artificial Intelligence*, May 2009 vol. 56, pp. 23–42. Berlin, Germany: Springer.

[Dixon, Duncan, and Kissinger, 2010] Dixon, Lucas, Duncan, Ross, and Kissinger, Aleks. (2010) 'Open Graphs and Computational Reasoning', *Proceedings of the Sixth Workshop on Developments in Computational Models: Causality, Computation, and Physics.* Waterloo, New South Wales: Electronic Proceedings in Theoretical Computer Science.

[Dixon and Kissinger, 2013] Dixon, Lucas and Kissinger, Aleks. (2013) 'Open Graphs and Monoidal Theories', *Mathematical Structures in Computer Science* vol. 23, no. 2, pp. 308–359. Cambridge, England: Cambridge University Press.

[Duncan and Lucas, 2013] Duncan, Ross and Lucas, Maxime. (2013) 'Verifying the Steane code with Quantomatic', *Proceedings of the 10th International Workshop on Quantum Physics and Logic (QPL).* Waterloo, New South Wales: Electronic Proceedings in Theoretical Computer Science.

[Eppstein, 2013] Eppstein, David. (2013) 'Which subgraphs have polynomially many connected subgraphs?', *11011110.*

`https://11011110.github.io/blog/2013/01/26/which-graphs-have.html`. Accessed November 2018.

[Gamma et al., 1994] Gamma, Erich, Helm, Richard, Johnson, Ralph, and Vlissides, John. (1994) *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley: Boston, Massachusetts.

[Garvie and Duncan, 2017] Garvie, Liam and Duncan, Ross. (2017) 'Verifying the smallest interesting colour code with Quantomatic', original preprint edition arXiv:1706.02717v1. Later abridged for publication in *Proceedings of the 14th International Workshop on Quantum Physics and Logic (QPL).* Waterloo, New South Wales: Electronic Proceedings in Theoretical Computer Science.

[Hadzihasanovic, 2015] Hadzihasanovic, Amar. (2015) 'A Diagrammatic Axiomatisation for Qubit Entanglement', *Proceedings of the 2015 30th Annual ACM/IEEE Symposium on Logic in Computer Science.* Washington DC: IEEE Computer Society.

[Hu, 2006] Hu, Yifan. (2006) 'Efficient, High-Quality Force Directed Graph Drawing', *The Mathematica Journal* vol. 10, no. 1, pp. 37–71. Champaign, Illinois: Wolfram Media.

[Kaufmann and Wagner (eds.), 2001] Kaufmann, Michael and Wagner, Dorothea (eds.). (2001) *Lecture Notes in Computer Science 2025: Drawing Graphs: Methods and Models.* Berlin, Germany: Springer-Verlag.

[Kissinger, 2012] Kissinger, Aleks. (2012) *Pictures of Processes: Automated Graph Rewriting for Monoidal Categories and Applications to Quantum Computing.* Doctor of Philosophy Thesis, University of Oxford.

[Kissinger and Zamdzhiev, 2015] Kissinger, Aleks and Zamdzhiev, Vladimir. (2015) 'Quantomatic: A Proof Assistant for Diagrammatic Reasoning', *Proceedings of the 25th International Conference on Automated Deduction.* Berlin, Germany: Springer.

[Martin, 2007] Martin, Adam. (2007) 'Entity Systems are the future of MMOG development - Part 2', *T-machine.org.* `http://t-machine.org/index.php/2007/11/11/`

`entity-systems-are-the-future-of-mmog-development-part-2/`. Retrieved May 2019.

[Merry, 2013] Merry, Alexander. (2013) *Reasoning with !-Graphs.* Doctor of Philosophy Thesis, University of Oxford.

[Lack and Sobociński, 2004] Lack, Stephen and Sobociński, Paweł. (2004) 'Adhesive Categories', *Proceedings of the 7th International Conference in Foundations of Software Science and Computation Structures, held as part of the Joint European Conferences on Theory and Practice of Software, March 29 – April 2 2004, Barcelona, Spain* pp. 273–288. Berlin, Germany: Springer.

[Rota, 1997] Rota, Gian-Carlo. (1997) 'Ten Lessons I Wish I Had Been Taught', *Notices of the American Mathematical Society* vol. 44 no. 1 pp. 22–25. Providence, Rhode Island: American Mathematical Society.

[Schöning, 1987] Schöning, Uwe. (1987) 'Graph isomorphism is in the low hierarchy', *Proceedings of the 4th Annual Symposium on Theoretical Aspects of Computer Science*, pp. 114–124. Berlin, Germany: Spinger-Verlag.

[Stefik and Hanenberg, 2017] Stefik, Andreas and Hanenberg, Stefan. (2017) 'Methodological Irregularities in Programming-Language Research', *Computer* vol. 50, no. 8, pp. 60–63. Washington, District of Columbia: Institute of Electrical and Electronics Engineers Computer Society.

[Williams, 2014] Williams, Virginia Vassilevska. (2014) 'Multiplying matrices faster than Coppersmith-Winograd', *Proceedings of the 44th annual ACM symposium on Theory Of Computing (STOC '12)*, pp. 887–898. Association for Computing Machinery: New York, New York.

[Breeze] Hall, David, Ramage, Daniel, et al. (far too many people to list here; see `https://github.com/scalanlp/breeze/blob/releases/v1.0-RC2/README.md`). *Breeze.* `http://www.scalanlp.org`. Accessed January 2019.

[Coq] Action for Technological Development Coq (far too many people to list here; see `https://coq.inria.fr/about-coq`, section ‘Credits’). *Coq.* `https://coq.inria.fr`. Accessed July 2018.

[Graphviz] Ellson, John, Gansner, Emden, Hu, Yifan, Janssen, Erwin, and North, Stephen. *Graphviz.* `https://www.graphviz.org`. Accessed July 2018. **See also** Ellson, John, Gansner, Emden R., Koutsofios, Eleftherios, North, Stephen C., and Woodhull, Gordon. (2004) ‘Graphviz and Dynagraph — Static and Dynamic Graph Drawing Tools’. In: Jünger, Michael and Mutzel, Petra (eds.). *Graph Drawing Software* pp. 127–148. Berlin, Germany: Springer-Verlag.

[Isabelle] Paulson, Lawrence C. et al. (far too many people to list here; see `https://isabelle.in.tum.de/dist/Isabelle2017/CONTRIBUTORS`). *Isabelle.* `https://isabelle.in.tum.de`. Accessed July 2018.

[Matita] Asperti, Andrea, Guidi, Ferruccio, Ricciotti, Wilmer, Coen, Claudio Sacerdoti, Tassi, Enrico, Zacchiroli, Stefano, Padovani, Luca, Schena, Irene, Di Lena, Pietro, Galatá, Michele, Griggio, Alberto, Selmi, Matteo, and Tamburreli, Vincenzo. *Matita.* `http://matita.cs.unibo.it`. Accessed July 2018.

[Mizar] Trybulec, Andrzej, Bylinski, Czeslaw, Grabowski, Adam, Kornilowicz, Artur, Naumowicz, Aam, Pąk, Karol, Urban, Josef, Bancerek, Grzegorz, and Milewski, Robert. *Mizar.* `http://mizar.org`. Accessed July 2018.

[NuPRL] Constable, R. L., Allen, S. F., Bromley, H. M., Cleaveland, W. R., Cremer, J. F., Harper, R. W., Howe, D. J., Knoblock, T. B., Mendler, N. P., Panangaden, P., Sasaki, J. T., and Smith, S. F. *Implementing Mathematics with the NuPRL Proof Development System.* `http://www.nuprl.org/book`. Accessed July 2018.

[Quantomatic] Kissinger, Aleks, Merry, Alex, Frot, Ben, Coecke, Bob, Quick, David, Miller-Bakewell, Hector, Dixon, Lucas, Soloviev, Matvey, Duncan, Ross, and Zamdzhiev, Vladimir. *Quantomatic.* `https://quantomatic.github.io/`. Accessed June 2018.