Open Graphs and Computational Reasoning

Lucas Dixon
University of Edinburgh
l.dixon@ed.ac.uk

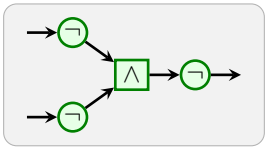
Ross Duncan, Aleks Kissinger
University of Oxford
{ross.duncan, alexander.kissinger}@comlab.ox.ac.uk

We present a form of algebraic reasoning for computational objects which are expressed as graphs. Edges describe the flow of data between primitive operations which are represented by vertices. These graphs have an interface made of half-edges (edges which are drawn with an unconnected end) and enjoy rich compositional principles by connecting graphs along these half-edges. In particular, this allows equations and rewrite rules to be specified between graphs. Particular computational models can then be encoded as an axiomatic set of such rules. Further rules can be derived graphically and rewriting can be used to simulate the dynamics of a computational system, e.g. evaluating a program on an input. Examples of models which can be formalised in this way include traditional electronic circuits as well as recent categorical accounts of quantum information.

1 Introduction

Graphs provide a rich language for specification and reasoning. Well known examples include the proof-nets of linear logic [6], Penrose's tensor notation [13], Feynman diagrams, and the common diagrams used for electronic circuits. Recently, graphs have also been used to formalise molecular biology [3] and quantum information processing [2].

This paper presents *open graphs* as a formal foundation for reasoning about computational structures. These graphs have a directed boundary, visualised as edges entering or leaving the graph. The boundary of a graph represents the inputs and outputs of the computation. This lets graphs be interpreted as compound computations with vertices as the atomic operations. For example, an electronic circuit that defines the compound logical operation of an or-gate, using not-gates around an and-gate, can be drawn as:



The structure and dynamics of a computational model are written in our formalism as a set of axiomatic rules between graphs. These graphical rules are declarative in the sense that the graphs they involve can be rewritten to derive new graphical rules in a conservative manner. In this way, our formalism is a logical framework to describe models of computation and derive new results. Another interesting feature of the formalism is that it provides a clear distinction between sequential and parallel dynamics in terms of composing rewrites using the underlying compositional principles of graphs.

Our main contribution is a concise formalism that provides graphs with a rich compositional structure and a convenient algebraic language. The formalism includes graphical concepts of addition, subtraction, tensor product, and substitution, with familiar laws. Building on these definitions, we develop the mathematics to support declarative graphical equations and directed rewrites between graphs. In particular, our development supports rewriting graphs that contain cycles edges. Care has also been taken

S. B. Cooper, E. Kashefi, P. Panangaden (Eds.): Developments in Computational Models (DCM 2010)
EPTCS 26, 2010, pp. 169–180, doi:10.4204/EPTCS.26.16

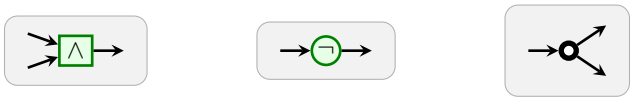
to provide a presentation that can be directly implemented as a software tool. The intention is to allow techniques from automated reasoning, such as Knuth-Bendix completation [7], to be employed. Thus graph matching directly incorperates associativity and commutativity. Another notable feature of the formalism is that it has a direct correspondence to its visualisation. In this paper, we simply state the properties of the formalism, leaving the proofs for a longer manuscript.

For the sake of conciseness and understandability, we illustrate our formalism with boolean logic circuits. More generally, our formalism can describe a variety of computational models, including categorical models of quantum information [4].

The paper is structured as follows: In §2, we introduce an informal example of reasoning with graphs that model boolean logic in electronic circuits. This introduces the key challenges. In §3, we place our contribution in the context of related work. We introduce our formalism by defining open graphs, in §4. We then discuss how these graphs compared and how one can be found within another in §5. Composition of graphs and rewriting are described in §6. In §7 we introduce the composition of rewrites and how computational models can be encoded as graphical theories. We conclude and discuss future work in §8.

2 Motivating example: electronic circuits

To motivate our formalism, we introduce a graph-based representation for boolean circuits. We start by giving its generating graphs and defining axioms. The basic generators for electronic circuits are the following:

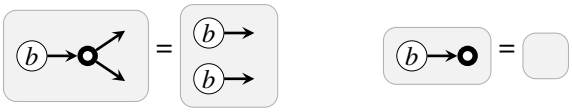


Any circuit can be built from these components by connecting them together along the half-edges. The unconnected half-edges that enter a circuit are the inputs, and those that leave are the outputs. Notice that copying a value is an explicit operation.

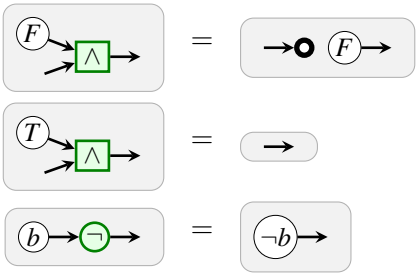
These circuits have two kinds of unitary-generators for the inputs. There are two unit-generators for the two boolean values and one counit-generator for when an output is ignored. These are drawn as:

where we write b for a boolean value which can either be F for false or T for true.

We can now describe circuits with some values given to them, and with wires that just stop. This is a simple but important class of problems. For instance, it includes satisfiability questions, which are formed by asking whether a given graph can be rewritten to the single input unit with value T. To answer such questions, and more generally to describe structural equivalences and the dynamics of boolean circuits, some axioms need to be introduced. For copying and ignoring values, these are:



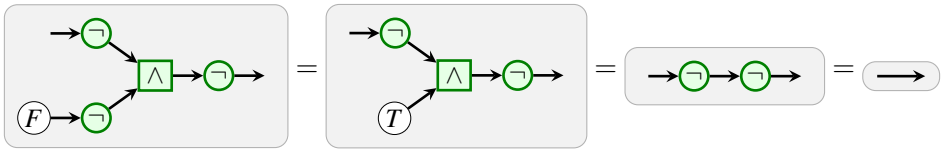
The axioms for conjunction (and-gates: $\wedge$) and negation (not-gates: $\neg$) are:



These provide a sufficient description for evaluating boolean circuits. Applying these axioms from left to right simulates evaluation and also performs simplification. The validity of axioms can be verified by checking the truth tables.

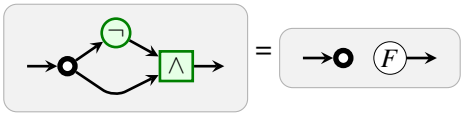
Although the above rules are sufficient for evaluation (when a circuit has all inputs given), they cannot prove all true equations about boolean circuits. To get a complete set of equations, new graphical axioms need to be introduced. For instance, we could easily verify that  .

The exhaustive analysis which is performed by examining truth tables can also be perfomed directly in the graphical language. We can examine every input to a graphical equation to see if the left and right hand sides evaluate to the same result. This corresponds to a proof by exhaustive case analysis. However, rules can also be derived directly by using the existing equations without examing all cases. For example, the equations above can be used with the evaluation axioms to carry out the following derivation:

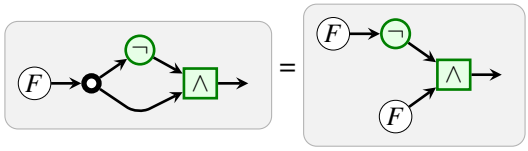


which proves that giving F to the compound or-gate is the same as the identiy on the other input. Such derivations can be exponentially shorter than case-analysis and, in the general setting, can be carried out in parallel when rewrite rules do not overlap. In the rest of this paper we focus on a formalism to support this kind of graphical reasoning.

Another salient feature of graph-based representations is that sharing and binding can be described using the graph's structure. For example, consider the following rule:



The graphical notation treats binding by the structure of edges. With a formula-based notation this could be described by an equation between lambda-terms $(\lambda x.\ ((\neg x) \wedge x)) = (\lambda x.\ F)$. This analogy results in composition along half-edges becoming function application of formula. For example plugging F into the left hand side of the equation could give the formula $(\lambda x.\ ((\neg x) \wedge x))F$. Beta-reduction, which reduces the formula to $(\neg F) \wedge F)$, corresponds to the copying rule. In the graphical language, the beta-reduction step is:



Notice that the graphical representation controls copying carefully: by explicit application of equational rules. This is an essential feature in graphical representations of quantum information, where copying can only happen in restricted situations.

These graphs introduce a particular challenge to formalising rewriting when a direct graph may contain cycles. This is also needed by the graphical formalisms of quantum information. To understand the issue, consider the following:

the graph  can be rewritten from left to right by the equation  .

This results in the graph with a circular edge and no vertices:  . Such graphs can also be constructed by connecting graphs made only of edges. These circles in a graph have a natural interpretation for computational models which interpret graphs be as linear transformations. These interpretations, which treat spacial-adjacency as the tensor product, treat circles as scalars. More generally, such graphs can be understood as traced monoidal categories.

The potential to introduce circles and allow composition along half-edges is the major challenge for formalisating graph transformation in this setting. The ability to formally represent such graphs as finite objects in a computer program, and use them as part of rewriting, is the main application that our formalism tackles.

3 Related Work

General notions of graph transformations [5, 1], do not directly provide a suitable basis for formal equational reasoning about computational objects. In particular, general graph transformations are not mono, and hence cannot express equations. Moreoever, the usual formalisms for graphs do not allow circular edges. Bigraphs provide another general formalism for graph rewriting, but they are significantly more complex [12]. Bigraphs use hyper-graphs, where our edges have a single source and target, and bigraphs also introduce a rich hierarchical structure.

Rewriting with graph-based presentations of computational systems has been studied with a variety of formalisms by Lafont [10, 11, 9, 8]. For instance, Operads and PROPs provide a notable way to rewrite graphical representations of composable, multi-input and multi-output functions. A wide variety monoidal categories (and higher-categories) also enjoy graphical representations [14]. Open graphs have a close correspondence to traced symmetric monoidal categories, but they absorb the braiding operations on tensor products. The main difference with our work is that we formalise graphs directly, rather than treat them as a presentation of an algebraic structure. This makes our formalism particularly amenable to software implementation [1]. Our notion of matching directly absorbs associative-commutative structure.

Work in systems biology provides another formalism for graphs that is similar to the one presented here [3]. The main difference is that our notion of matching is stricter: we require all incident edges to be specified in a rule. This is needed to ensure that a vertex has a fixed number of inputs and outputs.

In [4] we introduced a formalism for reasoning about categorical models of quantum information. We have since made several important improvements: matching and composition are now dual notions which allows a concise definition rewriting.

[1] For example, see our implementation of Quantomatic: `http://dream.inf.ed.ac.uk/projects/quantomatic`.

4 Open Graphs and Embeddings

Definition 4.1 (Pre-Open graph)**.** Let $\mathscr{G}$ be the following finite coproduct sketch.

$$E \underset{t}{\overset{s}{\rightrightarrows}} V + \varepsilon$$

We call models of $\mathscr{G}$ in **Set** *pre-open graphs*.

For a pre-open graph G, $G(V+\varepsilon)$ identifies a set of points, with V being the vertices, and ε being points on an edge, which we call *edge points*. If $G(\varepsilon) = \{\}$, we recover the usual notion of directed graph. Edge-points provide a combinatorial description of 'dangling' edges, 'half' edges, and edges attached to themselves (circles). In particular, they allow graphs to be composed by edge-points. Where convenient, we will use subscript notation to refer to elements in the model of $\mathscr{G}$, i.e. $s_G := G(s)$ and $E_G := G(E)$.

A pre-open graph is called an *open graph* if the edge points behave as if they were really points occurring within a single directed edge.

Definition 4.2 (Open graph)**.** For a pre-open graph G, let $s' : E_s \to \varepsilon$, $t' : E_t \to \varepsilon$ be the restrictions of s and t to edge points. G is called an *open graph* when s' and t' are injective. Let **OGraph** be the full subcategory of $Mod(\mathscr{G})$ whose objects are open graphs.

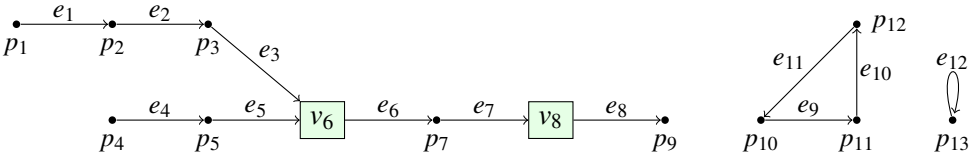
Injectivity ensures edges are not allowed to branch. Each point has at most one incoming edge (called an *in-edge*) and one outgoing edge (called an *out-edge*). Only vertices may have multiple in-edges and/or multiple out-edges.

Definitions 4.3. If a point $p \in \varepsilon_G$ has one out-edge, but no in-edges, it is called an *input*. Similarly, a point with one in-edge and no out-edges is called an *output*. The inputs and outputs define a graph's *boundary*. If a point has no in-edges and no out-edges, it is called an *isolated point*. We use the following notation for these subsets of ε_G:

1. $In(G)$ the set of inputs,
2. $Out(G)$ the set of outputs,
3. $Bound(G) = In(G) \cup Out(G)$ the set of boundary points, and
4. $Isol(G)$ the set of isolated points.

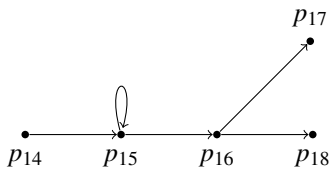
A graph consisting of only isolated points is called a *point-graph*. These will be used to define how graphs are composed.

Example 4.4. The following is an illustration of an open graph.



where p_i are edge points and v_i are vertices. Note that p_1, p_4 and p_8 are boundary points.

The following is not an open graph, because the map s is not injective for p_{15} and t is not injective for p_{15} and p_{16}



Definition 4.5. An *open embedding* is a monomorphism $e : G \to H$ between pre-open graphs that is full on vertices: for all p if $e(p) \in V_H$ then all edges adjacent to $e(p)$ are in the image of e.

Open embeddings describe how one graph can be found within another one and they will be used to identify the parts of a graph which can be rewritten by a rule. The identity map, the empty map, and graph isomorphisms are open embeddings.

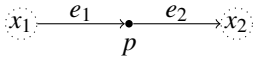
Example 4.6. The graph on the left has an open embedding into example 4.4, whereas the graph on the right does not.

The set of all open embeddings into an open graph, G, is analogous to the notion of open subsets of G, if G were to be considered a graph in the topological sense. The notion of open is a midway point between the usual combinatorial definition, and the idea of a graph as a topological space made by gluing together 1-dimensional manifolds.

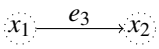
5 Homeomorphism and Matching

While edge points are a useful tool for composition, the number of edge points along an edge is irrelevant to the intended meaning of the graph. Just as two copies of the interval $[0,1]$ glued end to end are homeomorphic to just one copy, we shall define a concept of homeomorphic open graphs which is a course-graining of graph isomorphism.

Definition 5.1. We say that G *contracts* to H, if G can be made isomorphic with H by replacing any number of subgraphs of this form:



where $p \in \varepsilon$ and x_1, x_2 are (not necessary distinct) points in $V + \varepsilon$, with a graphs of this form:



In such a case, we write $H \preceq G$. Let $\sim$ then be the symmetric closure of $\preceq$. We say two graphs are *homeomorphic* iff $G \sim H$.

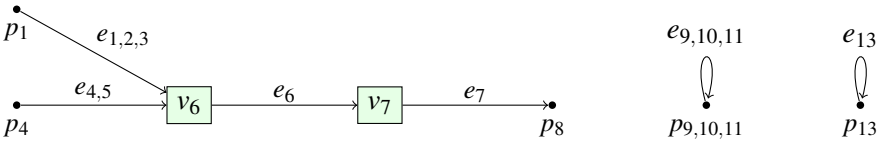
Proposition 5.2. *$\preceq$ is a well-founded, confluent partial order, up to graph isomorphism.*

Definition 5.3. An open graph is said to be *proper* if it is minimal with respect to $\preceq$.

Proposition 5.4. *Every graph has a unique minimal form. For graphs G, H, $G \sim H$ iff their minimal forms are isomorphic.*

Proper graphs are of particular interest for computing with open graphs because they provide a computable, minimal representations for open graphs.

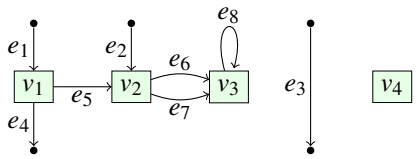
Example 5.5. The associated proper open graph of example 4.4 is:



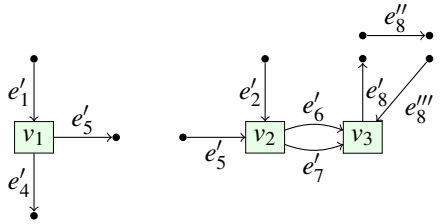
By considering open embeddings up to the relation $\sim$ we get a more general notion of what it means for a graph to be "inside" of another. This abstracts over any intermediate edge-points.

Definition 5.6. A graph L is said to *match* G, denoted by $L \tilde{\hookrightarrow} G$, if there exists some $G' \sim G$ and an open embedding $e : L \to G'$. Such an embedding is called a *matching* of L on G.

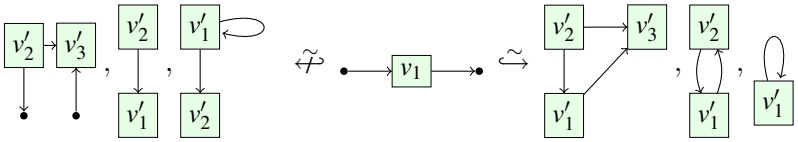
Example 5.7. Let G be the open graph:



then the following graph, H, matches G, where each edge of the form $e_i^{(n)}$ in H maps on to an edge expanded from the edge e_i in G.



Example 5.8. Some more examples of graphs and matchings. The graph in the centre matches those on its right, but not those on its left:



6 Composition and Rewriting

In this section, we define the notions of composing graphs along their boundaries and performing rewrites of graphs embedded inside of other graphs. First, we define a notion of boundary and interface for a graph.

Definition 6.1 (Boundary embedding)**.** A *boundary embedding* b is an open embedding from a point graph, P, to a graph, G, such that only boundary points are in the image of b: $\forall x \in \varepsilon_P.\ b(x) \in Bound(G)$.

Boundary embeddings identify a subset of a graph's boundary points along which the graph can be composed with another graph. In particular, they let us define the *interface* of a graph.

Definition 6.2 (Interface)**.** The *interface* of a graph G, written $Interf(G)$, is a tuple (P_I, b_I, P_O, b_O), where P_I and P_O are point graphs such that $b_I : P_I \to G$ is a boundary embedding that is surjective on $In(G)$, and $b_O : P_O \to G$ is a boundary embedding that is surjective $Out(G)$.

Definition 6.3. Two graphs G and G' are said to have *the same boundary* (or *the same interface*) when $Interf(G)$ and $Interf(G')$ are isomorphic.

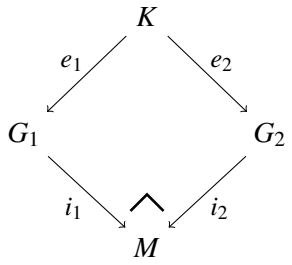
Colimits can be used to 'glue' multiple graphs together. Informally, they define minimal graphs that a given set of other graphs can be embedded into. We will define *open colimits* as colimits with conditions to ensure that they merge graphs coherently. This will provide a notion of merged graph that is coherent with its components, and in particular, with which we can define composition along half-edges. Whereas the category $Mod(\mathscr{G})$ has all small colimits, **OGraph** only has certain small colimits, namely those that will not break the injectivity condition on the edge points.

Definition 6.4. A pair of arrows $f : G \to H_1$, $g : G \to H_2$ in **OGraph** are said to be *boundary-coherent* when

1. for all $p \in Bound(G)$, $f(p) \notin Bound(H_1) \implies g(p) \in Bound(H_2)$, and
2. for all $p \in Isol(G)$, either $f(p)$ is an isolated point, $g(p)$ is an isolated point, or $f(p)$ and $g(p)$ are compatible boundaries. That is, one is an input iff the other is an output.

Definition 6.5. A colimit of a diagram D in **OGraph** is called an *open colimit* when all pairs of arrows in D are boundary-coherent.

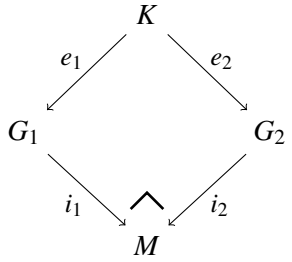
We will focus on the notion of open pushouts, which we shall call *mergings*, that arrises from open colimits. In particular, given graphs G_1, G_2 and K, with open embeddings $e_1 : K \hookrightarrow G_1$ and $e_2 : K \hookrightarrow G_2$, a graph M is called the *merging* of G_1 and G_2 (on K by e_1 and e_2) when it is defined by the pushout:



This makes M the smallest graph containing G_1 and G_2 merged exactly on the graph K, as identified by e_1 and e_2. Given $p := (K, e_1, e_2)$, we use the notation $G_1 +_p G_2$ to denote M. If the open embeddings associated with pushout triples p and q are disjoint, we can take the merging to be strictly associative and omit parentheses: $G_1 +_p G_2 +_q G_3 := (G_1 +_p G_2) +_q G_3 = G_1 +_p (G_2 +_q G_3)$

Theorem 6.6. ***OGraph*** *has all small open colimits. Furthermore, colimit maps are full on vertices.*

Corollary 6.7. *Given a merging,*



the maps i_1, i_2 are open embeddings.

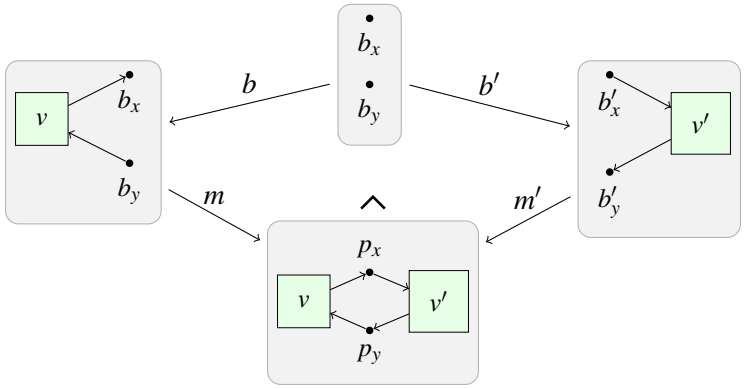
This ensures that the visual intuition of finding one graph in another is preserved: merging graphs does not change their internal structure and hence we can find a graph G (an open embedding of it) within a any merging of G with any other graph.

Definition 6.8 (Tensor)**.** When the graph being merged-on is empty, we call the resulting merged graph, $G +_{(\emptyset,\emptyset,\emptyset)} H$, the *tensor product*, and write it as $G \otimes H$.

Tensor composition corresponds to placing graphs side by side. To define composition along half-edges, we use point graphs and a special kind of open embedding that identifies (part of) the boundary.

Definition 6.9 (Plugging)**.** A graph merging $G +_{(P,b,b')} G'$ is called a *plugging*, and is written $G +^*_{(P,b,b')} G'$, when P is a point graph and b and b' are boundary-coherent boundary embeddings.

Example 6.10. An example of plugging using pushouts. The grey boxes are drawn around the graphs involved to distinguish between edges in the graphs and those of the pushout diagram.



Proposition 6.11. *Plugging respects the equivalence class of homeomorphisms:* $G +^*_{(K,e_1,e_2)} H \sim G' +^*_{(K,e'_1,e'_2)} H'$ *iff* $G' \sim G$ *and* $H' \sim H$*, where* e'_1*,* e'_2 *are the embeddings* e_1 *and* e_2*, but considered as maps into* G' *and* H'*, respectively.*

This allows any definitions built by plugging graphs together to be lifted to the equivalence class induced by homeomorphism.

Proposition 6.12. *The most general plugging with respect to matching is* $\otimes$*:*

$$(H +^*_p G \overset{\sim}{\hookrightarrow} K) \Rightarrow (H \otimes G \overset{\sim}{\hookrightarrow} K)$$

Proposition 6.13. *Every open embedding* $e : G \hookrightarrow M$ *defines a unique graph* H *that is all of* M *except for the image of* e *(upto the boundary points). In particular, it identifies a unique* p *such that* $G +^*_p H = M$*.*

Proposition 6.14. *Every merging* $G +_{(K,e_1,e_1)} H$ *can be written as graphs* G' *and* H' *that are plugged onto* K *such that* $G +_{(K,e_1,e_1)} H = G' +^*_p K +^*_q H'$*,* $G = G' +^*_p K$*,* $H = K +^*_q H'$*.*

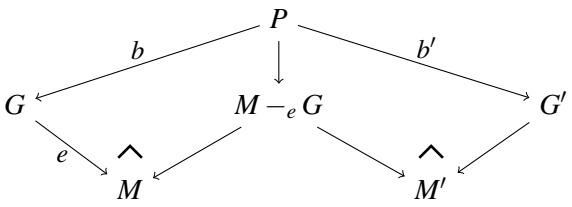
This, combined with plugging respecting homeomorphism (proposition 6.11), allows merging (and any other derived notions) to also respect the homeomorphism.

Definition 6.15 (Subtraction)**.** We define the *subtraction* of G from M, at $e : G \hookrightarrow M$, written $M -_e G$, as the unique graph H such that $G +^*_p H = M$.

When the embedding is implicit, we write $M - G$.

Proposition 6.16. *Subtracts always embed:* $G - H \hookrightarrow G$

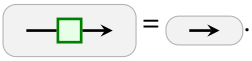
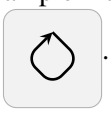
Definition 6.17 (Substitution)**.** A graph G occurring within M can be substituted, at $e : G \hookrightarrow M$, for another graph G', when G and G' share the same boundary (a subset of which is identified by P, b and b'). The resulting graph, M' is defined according to the following pair of pushouts:

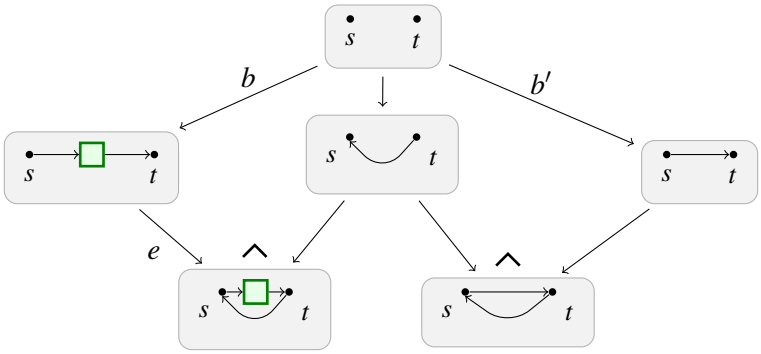


where the left pushout computes the subtraction $M -_e G$, and the right pushout then computes the plugging to form $(M -_e G) +_p G'$.

Analogously to traditional notation for substitution, we will write M' as $M[G \mapsto G']_e$, and when the embedding is implicit we write $M[G \mapsto G']$.

Proposition 6.18. *Substitution preserves the boundary: M and $M[G \mapsto G']$ have the same boundary.*

Example 6.19 (Circles). We can now returning to the challenging example introduced at the end of § 2. We will rewrite the graph  by  to get  . The pushout construction for this substitution is as follows:



Definition 6.20. A *rewrite*, r, is a pair of graphs, (L,R), with the same boundary.

We will denote that (L,R) is a rewrite named r by $r : L \to R$.

Definition 6.21. *Applying a rewrite $r : L \to R$* to a graph G at a *match* $e : L \hookrightarrow G'$ (for $G' \sim G$) is said to *rewrite* G to H, when $H = G'[L \mapsto R]_e$. The induced rewrite $G \to H$ is called an *extension* of r, and named $r^{\uparrow e}$.

7 Graphical Theories

We will now develop a categorical description of *graphical theories*. This allows the usual concepts from rewriting theory, such as normalisation, termination, confluence, etc. to be employed. However, rather than concern ourselves with these familiar concepts, we will focus on the structure of graphical theories in terms of how rules can be combined, sequentially and in parallel, using the underlying graph operations.

Definition 7.1 (Sequential Merging of Rewrites)**.** Given rewrite rules $r : L \to R$ and $r' : L' \to R'$, and a merged graph $M := R +_p L'$, with $p := (K, e, e')$ and open embeddings $m : R \hookrightarrow M$ and $m' : L' \hookrightarrow M$ induced by the pushout, then the *sequential merging* of r and r' at p defines the new rewrite rule:

$$(r ;_p r') : (M[R \mapsto L]_m) \to (M[L' \mapsto R']_{m'})$$

Sequential merging does nothing more than rewrites with extension:

Proposition 7.2 (Soundness)**.** *if $(r_1 ;_p r_2) : G \to G'$ is a rewrite; then there exists a graph H, and embeddings e_1 and e_2 such that $G \xrightarrow{r_1{}^{\uparrow e_1}} H \xrightarrow{r_2{}^{\uparrow e_2}} G'$.*

While merging provides sequential composition, lifting the concept of graph plugging to rewrites provides a notion of parallel composition.

Proposition 7.3. *Given rewrite rules $r : L \to R$ and $r' : L' \to R'$; then $L +^*_p L'$ has the same boundary as $R +^*_p R'$.*

Proposition 7.4 (Completeness)**.** *if $M \xrightarrow{r_1{\uparrow}e_1} M_2 \xrightarrow{r_2{\uparrow}e_2} M_3$ then there exists an e' and p such that $(r_1 ;_p r_2)^{\uparrow e'} : M \longrightarrow M_3$*

Completeness can also be thought of as 'compressing' any sequence of rewrites into a single larger rewrite $(r_1 ;_p r_2)$ that does all the steps at once.

Definition 7.5 (Parallel composition of rewrites)**.** Given rewrites $r : L \to R$ and $r' : L' \to R'$, the *plugging of rewrites*, also called the *parallel composition of rewrites*, is defined as

$$(r_1 +^*_p r_2) : (L_1 +^*_p L_2) \to (R_1 +^*_p R_2)$$

Proposition 7.6. *Plugging is a special case of merging: for every plugging of rewrites p, the sequential merging produces the same rewrite $r_1 +^*_p r_2 = r_1 ;_q r_2$*

A special case of plugging rewrites together is the tensor product of two rewrites: $r \otimes r' := (L \otimes L', R \otimes R')$.

Proposition 7.7. *If $(r_1 \otimes r_2) : G \to M$ is a rewrite then there exists an H and H' such that*

$$G \xrightarrow{r_1} H \xrightarrow{r_2} M \qquad \text{and} \qquad G \xrightarrow{r_2} H' \xrightarrow{r_1} M$$

Proposition 7.8. *The most general parallel composition of rewrites is the tensor product. Given $(r +^*_p r')^{\uparrow e_1} : G \to H$ then there exists e_2 such that $(r \otimes r')^{\uparrow e_2} : G \to H$*

We now observe that the extension of a rewrite rule can also be understood in terms of plugging.

Proposition 7.9. *Given a rewrite rule $r : G \to G'$, and an extension of it $r' : M \to M'$, then there is an identity rule $\mathrm{id}_H : H \to H$, such that $r' = r +^*_p \mathrm{id}_H$ for some p.*

Definition 7.10 (Graphical Theory)**.** Given $\Gamma := (S, R)$, where S is a *generating set of graphs*, closed under plugging, and a R is a *generating set of rewrites*, closed under extension and formed from S; the *graphical theory*, $\mathbf{GThy}(\Gamma)$ is the category with S as the objects and arrows formed by formal sequential compositions of rewrites in R. Each arrow is defined by a particular composition. The generating set R is also called the *rewrite rules* of the theory, while compositions are called *rewrite sequences*.

Proposition 7.11. *Every graphical theory is a monoidal category, with tensor product on objects inherited from **OGraph**, and on rewrites as the special case of plugging described above.*

A model of computation can now be characterised by a particular graphical theory. The generating graphs describe the objects of interest and axioms are the generating rewrites rules which describe the model's interesting structural dynamics. For instance, the example introduced in §2 is a graphical theory.

A direct result of proposition7.6 is that graphical theories also have parallel and tensor composition. A consequence of soundness (proposition 7.2) for graphical theories is that, given an initial set of rules, new rules in the theory may be safely derived by sequential merging of existing ones; these new rules will also be in the graphical theory. This gives an algorithm to derive new rules from existing ones which are treated as axioms of a computational model.

8 Conclusions and Further Work

We have formalised a compositional account of graphs which represent computational processes. These graphs have an interface made of half-edges that enter or leave the graph. Methods to support graphical rewriting have been described, and it has been shown how graphical rewriting rules can themselves be composed. The construction is by an richer intermediate notion of graphs with edges-points. These more exotic structures provide the needed structure for composition by pushouts. In particular, they allow rewriting to preserve a graphs interface, even in the presence of cycles.

This foundation allows a variety of techniques from rewriting, such as Knuth-Bendix completion [7], to be lifted to reasoning about computational graphs. Another area of further work is to extending this formalism to include bang-boxes, as introduced in [4], as well as other kind of iterative and recursive structure. We have started to study the categorical structure of graphical theories, but there is a lot more to be considered, such as the relationship between graphical theories and the category of open graphs. The exact of the relationships between open graphs, topological analogies, and other graphical formalisms is also important future research.

Acknowledgements This research was funded by EPSRC grants EPE/005713/1 and EP/E04006/1, and by a Clarendon Studentship. Thanks also to Jeff Egger and Rod Burstall for their helpful discussions.

References

[1] P. Baldan, A. Corradini, and B. König. Unfolding graph transformation systems: Theory and applications to verification. pages 16–36, 2008.

[2] B. Coecke and R. Duncan. Interacting quantum observables. In *ICALP 2008*. LNCS, 2008.

[3] V. Danos and C. Laneve. Formal molecular biology. *Theoretical Computer Science*, 325(1):69–110, 2004.

[4] L. Dixon and R. Duncan. Graphical reasoning in compact closed categories for quantum computation. *AMAI*, 56(1):20, 2009.

[5] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science. EATCS Series)*. Springer, 2006.

[6] J.-Y. Girard. Proof-nets: The parallel syntax for proof-theory. In *Logic and Algebra*, pages 97–124. Marcel Dekker, 1996.

[7] D. E. Knuth and P. B. Bendix. Simple word problems in universal algebras. In *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon Press, 1970.

[8] Y. Lafont. Interaction nets. In *POPL '90: Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 95–108, New York, NY, USA, 1990. ACM.

[9] Y. Lafont. Towards an algebraic theory of boolean circuits. *Journal of Pure and Applied Algebra*, 184(2-3):257 – 310, 2003.

[10] Y. Lafont. Diagram rewriting and operads, 2010. In Lecture Notes from the Thematic school : Operads CIRM, Luminy (Marseille), 20-25 April 2009.

[11] Y. Lafont and P. Rannou. Diagram rewriting for orthogonal matrices: A study of critical peaks. In *RTA'08*, pages 232–245, Berlin, Heidelberg, 2008. Springer-Verlag.

[12] R. Milner. Pure bigraphs: Structure and dynamics. *Information and computation*, 204(1):60–122, 2006.

[13] R. Penrose. Applications of negative dimensional tensors. In *Combinatorial Mathematics and its Applications*, pages 221–244. Academic Press, 1971.

[14] P. Selinger. A survey of graphical languages for monoidal categories. *New Structures for Physics*, pages 275–337, 2009.