Notes on a formal theory of graphs

Lucius Gregory Meredith[1]

Managing Partner, RTech Unchained 9336 California Ave SW, Seattle, WA 98103, USA
lgreg.meredith@gmail.com

Abstract. We describe a formal theory of graphs with good complexity properties stemming from its compositional structure.

We describe an algebraic theory of graphs, $\mathsf{G}[X, V]$. The theory is dependent on a theory of variables, X, and a theory of vertices, V. Each theory is required to provide an effective procedure for deciding membership. More specifically, given any u we can decide $u \in X$ (and $u \in V$, respectively). Likewise, both dependencies must come with an effective notion of equality. That is, for variables x_1 and x_2 there is an effective procedure for deciding whether $x_1 = x_2$, and likewise for vertices. We refer to the graphs captured by the theory as the domain of graphs of $\mathsf{G}[X, V]$. The theory is unique in that it explicitly includes a notion of a reference to a vertex and only admits edges between references [1] [3] [2] [4]. Indeed, variables supply the references. We write

$$\mathsf{G}[X, V]; \Gamma \vdash g$$

to mean that – given the references Γ – g is well formed in $\mathsf{G}[X, V]$ and we use type inference rules of the form

$$\frac{\mathsf{G}[X, V]; \Gamma_1 \vdash g_1 \;\ldots\; \mathsf{G}[X, V]; \Gamma_n \vdash g_n}{\mathsf{G}[X, V]; K_\Gamma(\Gamma_1, \ldots, \Gamma_n) \vdash K_g(g_1, \ldots, g_n)} K_{ctor}$$

to indicate that given dependencies $K_\Gamma(\Gamma_1, \ldots, \Gamma_n)$ the graph, $K_g(g_1, \ldots, g_n)$, constructed from well-formed graphs $g_1, \ldots, g_n$, using the constructor K_g is well-formed. The effective procedure (aka algorithm) for determining when a graph is well formed is given by a collection of such rules.

There are two principal reasons for the development of this theory. One is complexity management. Whether they are the objects of investigations themselves, as in graph theory, or are being used as tools to support some other investigation such as in biology or physics or cryptocurrency, graphs are used primarily in *computations.* As such they should be *algorithmically* specified. The standard presentation of graphs as collections of vertices and edges does not treat them as algorithmically specified. To understand this lacunae consider the standard presentation of a linked list.

$$\mathsf{List}[A] = 1 + A \times \mathsf{List}[A]$$

This recursive specification matches the standard interpretation of lists containing elements of type A as either empty (the 1 in the domain equation) or formed by consing an element of type A onto a list containing elements of type A (the product $A \times \mathsf{List}[A]$). The recursive specification recognizes that lists are recursively built up from *lists* and a small set of operations.

Such an algebraic characterization of linked lists is typically used in algorithmic specifications of calculations involving lists. By contrast graphs are typically presented as a pair of collections of vertices and edges, rather than as a recursive data type involving operations that build graphs from graphs. This conception permeates many tools and software libraries that provide computational support for calculating with graphs and result in nearly overwhelming complexity.

An excellent example is the popular tool GraphViz and its dotty language. Using dotty to specify the complete graph of 100 nodes is an enormous task. The recursive data type resulting from the theory of graphs presented here results in a single line specification of the complete graph. More generally, in the examples we present we see an exponential jump in complexity in the types of graphs being characterized and yet the algorithmic characterizations remain typical 1-line recursive specifications.

The other motivation is foundational We seek a minimal characterization of subtypes of of the type of graphs.

0.1 Well-formedness

This is an algebraic theory, and thus it is syntactic in nature. It provides a language of graphs that are built out of logical sentences. In more detail, the theory admits three kinds of sentences:

- recognizing an admissible vertex: $\mathsf{G}[X, V]; \Gamma \vdash v$;
- recognizing an admissible variable: $\mathsf{G}[X, V]; \Gamma \vdash x$;
- and, recognizing a well formed graph: $\mathsf{G}[X, V]; \Gamma \vdash g$

Variables are used to capture references to vertices which are in turn used to form edges between vertices. As such, judging the wellformedness of a graph depends on the use of references. Hence, a judgement makes use of a dependency list, Γ which is just a sequence of variables. That is,

$$\Gamma ::= () \mid x, \Gamma$$

Notationally, we overload the comma to indicate concatenation of sequences. Thus, given $\Gamma_1 = x_{11}, \ldots, x_{1m}$ and $\Gamma_2 = x_{21}, \ldots, x_{2n}$, then $\Gamma_1, \Gamma_2 = x_{11}, \ldots, x_{1m}, x_{21}, \ldots, x_{2n}$

A graph expression is given by the grammar

$$g, h ::= 0 \mid v|g \mid g \otimes h \mid \mathsf{let}\ x = v\ \mathsf{in}\ g \mid \langle \mathsf{let}\ x_1 = v_1\ \mathsf{in}\ g, \mathsf{let}\ x_2 = v_2\ \mathsf{in}\ h \rangle$$

Pronunciation The graph constructors are pronounced as follows.

- 0 – "the empty graph", or just "empty";
- $v|g$ – "g with the vertex v adjoined", or "adjoin v to g";

- $g_1 \otimes g_2$ – "the graph formed by juxtaposing g_1 and g_2", or "juxtapose g_1 and g_2", or just "g_1 and g_2";
- $\mathsf{let}\ x = v\ \mathsf{in}\ g$ – "let x stand for v in g"; or "let x be v in g";
- $\langle \mathsf{let}\ x_1 = v_1\ \mathsf{in}\ g_1, \mathsf{let}\ x_2 = v_2\ \mathsf{in}\ g_2 \rangle$ – "the graph formed by connecting g_1 to g_2 with and edge from x_1 to x_2"; or, "connect g_1 to g_2 with and edge from x_1 to x_2".

The judgment $\mathsf{G}[X, V]; \Gamma \vdash g$ is pronounced "$\mathsf{G}[X, V]$ thinks that g is well-formed, given dependencies, Γ ." Similarly, $\mathsf{G}[X, V]; \Gamma \vdash v$ is pronounced "$\mathsf{G}[X, V]$ thinks that v is an admissible vertex, given dependencies, Γ;" and $\mathsf{G}[X, V]; \Gamma \vdash x$ is pronounced "$\mathsf{G}[X, V]$ thinks that x is an admissible variable, given dependencies, Γ."

A rule of the form

$$\frac{H_1, \ldots, H_n}{C} R$$

is pronounced "R concludes that C given H_1, …, H_n".

The rules for judging when a vertex or a variable are admissible or graph is well formed are as follows

$$\frac{}{\mathsf{G}[X, V]; () \vdash 0} Foundation$$

$$\frac{v \in V}{\mathsf{G}[X, V]; () \vdash v} Verticity \quad \frac{x \in X}{\mathsf{G}[X, V]; \emptyset \vdash x} Variation$$

$$\frac{\mathsf{G}[X, V]; \Gamma \vdash g \quad \mathsf{G}[X, V]; () \vdash v}{\mathsf{G}[X, V]; \Gamma \vdash v|g} Participation \quad \frac{\mathsf{G}[X, V]; \Gamma \vdash g \quad \mathsf{G}[X, V]; () \vdash x}{\mathsf{G}[X, V]; \Gamma, x \vdash x|g} Dependence$$

$$\frac{\mathsf{G}[X, V]; \Gamma_1 \vdash g_1 \ \mathsf{G}[X, V]; \Gamma_2 \vdash g_2}{\mathsf{G}[X, V]; \Gamma_1, \Gamma_2 \vdash g_1 \otimes g_2} Juxtaposition[\Gamma_1 \cap \Gamma_2 = \emptyset]$$

$$\frac{\mathsf{G}[X, V]; \Gamma, x \vdash g \quad \mathsf{G}[X, V]; () \vdash v}{\mathsf{G}[X, V]; \Gamma \vdash \mathsf{let}\ x = v\ \mathsf{in}\ g} Nomination$$

$$\frac{\mathsf{G}[X, V]; \Gamma_1 \vdash \mathsf{let}\ x_1 = v_1\ \mathsf{in}\ g_1 \quad \mathsf{G}[X, V]; \Gamma_2 \vdash \mathsf{let}\ x_2 = v_2\ \mathsf{in}\ g_2}{\mathsf{G}[X, V]; \Gamma_1, \Gamma_2 \vdash \langle \mathsf{let}\ x_1 = v_1\ \mathsf{in}\ g_1, \mathsf{let}\ x_2 = v_2\ \mathsf{in}\ g_2 \rangle} Connection[\Gamma_1 \cap \Gamma_2 = \emptyset]$$

Interpretation

- The *Foundation* rule says that regardless of the theory of vertices or variables, the empty graph is always well formed.
- The rules *Verticity* and *Variation* say that admissibility derives from the effective notion of membership required of the theory of vertices and the theory variables.
- The rule for *Participation* says that if $\mathsf{G}[X, V]$ thinks that g is well formed, given Γ, and $\mathsf{G}[X, V]$ thinks that v is admissible as a vertex, then $\mathsf{G}[X, V]$ thinks that adjoining v to g, i.e. $v|g$, is well formed.

- The rule for *Dependence* is similar to *Participation*. It says that $\mathsf{G}[X, V]$ thinks that it can adjoin x to g, i.e. $x|g$ to get a well formed graph given Γ if $\mathsf{G}[X, V]$ thinks that g is well formed, given Γ, x.
- The rule for *Juxtaposition* says that simply juxtaposing two well formed graphs results in a well formed graph. A careful accounting will observe that the dependencies of the two graphs must be disjoint.
- Logicians and computer scientists will recognize the rule for *Nomination* as a kind of cut rule. It's like β-reduction with an explicit substitution. Specifically, supposing that $\mathsf{G}[X, V]$ thinks that g is well formed, given Γ, x and that $\mathsf{G}[X, V]$ thinks that v is an admissible vertex then $\mathsf{G}[X, V]$ thinks that letting x stand for v in g is well formed, given Γ. A graph given by an expression of this form is said to be in nominated form.
- Finally, the rule for *Connection* says that an edge is only admissible between two well formed graphs in nominated form and then only when the dependencies are disjoint.

Notice that the intuitive interpretation for *Nomination* matches the pronunciation.

$$\frac{\mathsf{G}[X, V]; \Gamma, x \vdash g \quad \mathsf{G}[X, V]; () \vdash v}{\mathsf{G}[X, V]; \Gamma \vdash \mathsf{let}\ x = v\ \mathsf{in}\ g} Nomination$$

"Nomination concludes that $\mathsf{G}[X, V]$ thinks that the graph that lets x stand for v in g is well formed given Γ if $\mathsf{G}[X, V]$ thinks that g is well formed given Γ, x, and $\mathsf{G}[X, V]$ thinks that v is an admissible vertex."

For clarity we will often write $v|0$ as $[v]$.

0.2 Membership

We can extend the effective membership relation provided by V to one on $\mathsf{G}[X, V]$. Written $\mathsf{G}[X, V] \vdash v \in g$, the algorithm is given by the following set of rules

$$\frac{}{\mathsf{G}[X, V] \vdash v \in v|g} Ground$$

$$\frac{\mathsf{G}[X, V] \vdash v \in g}{\mathsf{G}[X, V] \vdash v \in g \otimes g'} Union$$

$$\frac{\mathsf{G}[X, V] \vdash v \in g}{\mathsf{G}[X, V] \vdash v \in \mathsf{let}\ x = v\ \mathsf{in}\ g} Transparency$$

$$\frac{\mathsf{G}[X, V] \vdash v \in g_1}{\mathsf{G}[X, V] \vdash v \in \langle \mathsf{let}\ x_1 = v_1\ \mathsf{in}\ g_1, \mathsf{let}\ x_2 = v_2\ \mathsf{in}\ g_2 \rangle} Link_L$$

$$\frac{\mathsf{G}[X, V] \vdash v \in g_2}{\mathsf{G}[X, V] \vdash v \in \langle \mathsf{let}\ x_1 = v_1\ \mathsf{in}\ g_1, \mathsf{let}\ x_2 = v_2\ \mathsf{in}\ g_2 \rangle} Link_R$$

As a matter of convenience we write $\mathsf{G}[X, V] \vdash v \notin g$ to mean that it is not the case that $\mathsf{G}[X, V] \vdash v \in g$. The inquisitive reader is invited to write a similar set of rules for edge membership. Of course, given a graph, g, it is very useful to refer to its collection of vertices and its collection of edges, which we denote $\mathsf{v}(g)$ and $\mathsf{e}(g)$ respectively.

0.3 Equations

The syntactic theory is too fine grained. It makes syntactic distinctions that do not correspond to distinct graphs. Juxtaposition is a great example. It corresponds to the disjoint sum of two graphs and as such the order of juxtaposition should not matter. We erase these syntactic distinctions with a set of equations on the graph expressions.

$$\dfrac{}{\mathsf{G}[X,V] \vdash 0 \otimes g = g}\,\mathit{Identity}$$

$$\dfrac{}{\mathsf{G}[X,V] \vdash g_1 \otimes g_2 = g_2 \otimes g_1}\,\mathit{Symmetry}$$

$$\dfrac{}{\mathsf{G}[X,V] \vdash g_1 \otimes (g_2 \otimes g_3) = (g_1 \otimes g_2) \otimes g_3}\,\mathit{Associativity}$$

$$\dfrac{}{\mathsf{G}[X,V] \vdash v_1|\ldots|v_i|v_j|g = v_1|\ldots|v_j|v_i|g}\,\mathit{Permutation}$$

$$\dfrac{}{\mathsf{G}[X,V] \vdash \mathsf{let}\ x = v\ \mathsf{in}\ \mathsf{let}\ x' = v\ \mathsf{in}\ g = \mathsf{let}\ x = v\ \mathsf{in}\ g}\,\mathit{Conservation}$$

$$\dfrac{\mathsf{G}[X,V] \vdash x \notin g_2}{\mathsf{G}[X,V] \vdash (\mathsf{let}\ x = v\ \mathsf{in}\ g_1) \otimes g_2 = \mathsf{let}\ x = v\ \mathsf{in}\ (g_1 \otimes g_2)}\,\mathit{Extrusion}$$

Note the mutual dependency amongst *Extrusion* and *Permutation* and the extended $\in$ relation.

1 Examples

1.1 Graphs as morphism

Let X be some set of variables, D be some domain, such as Nat, the natural numbers, or $Bool$, the booleans. We can describe many graphs as morphisms, $g(X, D) : D \to \mathsf{G}[X, D]$.

The discrete graph The discrete graph of n elements, written $\mathsf{discrete}(n)$, is simply n unconnected vertices. Noting that the graph consisting of a single vertex, n, has the form $n \to 0$, which adjoins n to the empty graph, 0, we can define the domain of discrete graphs recursively.

$\mathsf{discrete}(0) = 0$

$\mathsf{discrete}(n) = n|0 \otimes \mathsf{discrete}(n-1)$

$= [n] \otimes \mathsf{discrete}(n-1)$

Thus, $\mathsf{discrete}(n) = [n] \otimes [n-1] \otimes \cdots \otimes 0$

Domain elements as variables Sometimes it is convenient to use domain elements as variables as well as vertices; that is, it is often useful to form graphs from $\mathsf{G}[D, D]$. In this case, we distinguish the mention of a domain element, say d, in the role of a variable from the mention in the role of a vertex by quotation marks. Specifically, we write $\ulcorner d \urcorner$ when using d as a variable versus merely d when using d as a vertex. An example in context this might look like $\mathsf{let}\ \ulcorner d \urcorner = d\ \mathsf{in}\ g$.

The chain Using these two ideas we can give a recursive definition of the domain of chains. As with the discrete graph, the chain of 0 elements is the empty graph. The chain of a single element is not simply adjoining that element as a vertex to the empty graph. It also selects that element. Then the recursive specification creates an edge between the graph containing a single element, $n \to 0$, and the chain of $n-1$ elements by selecting the element, n and calling it $\ulcorner 2n-1 \urcorner$, and then linking the two resulting graphs. To ensure that the result is linkable it selects the element again, this time calling it $\ulcorner 2n \urcorner$.

$\mathsf{chain}(0) = 0$
$\mathsf{chain}(1) = \mathsf{let}\ \ulcorner 2 \urcorner = 1\ \mathsf{in}\ [1]$
$\mathsf{chain}(n) = \mathsf{let}\ \ulcorner 2n \urcorner = n\ \mathsf{in}\ (\mathsf{let}\ \ulcorner 2n-1 \urcorner = n\ \mathsf{in}\ [n], \mathsf{chain}(n-1))$
The reader can easily verify that for each $n \geq 1$ $\mathsf{chain}(n)$ is in nominated form.
Here is the chain from 2 to 1.

$$\mathsf{chain}(2) = \mathsf{let}\ \ulcorner 4 \urcorner = 2\ \mathsf{in}\ (\mathsf{let}\ \ulcorner 3 \urcorner = 2\ \mathsf{in}\ [2], \mathsf{let}\ \ulcorner 2 \urcorner = 1\ \mathsf{in}\ [1])$$

The cycle The cyclic graph of n vertices and n edges simply creates an additional edge from the end of the $\mathsf{chain}(n)$ back to the beginning.
$\mathsf{cycle}(0) = 0$
$\mathsf{cycle}(n) = \langle \mathsf{chain}(n), \mathsf{chain}(1) \rangle$

The complete graph This next example is made even more compact if we admit some syntactic sugar. First, let us stipulate that D supports a means of checking elements, such as d, for properties, say p, and write $p(d)$ for the predicate that determines when d inhabits the property p. We can lift the edge expression to an expression between collections of elements from D. Specifically, we can form

$$\langle \mathsf{for}(x_1 \leftarrow \mathsf{v}(g_1)\ \mathsf{if}\ p_1(x_1))\{g_1\}, \mathsf{for}(y \leftarrow \mathsf{v}(g_2)\ \mathsf{if}\ p_2(x_2))\{g_2\} \rangle$$

which adds to the graph $g_1 \otimes g_2$ an edge from each vertex in $\{v \in \mathsf{v}(g_1) | p_1(v)\}$ to each vertex in $\{v \in \mathsf{v}(g_2) | p_2(v)\}$

When $|\{v \in \mathsf{v}(g_i) | p_i(v)\}| = 1$ then this form is the same as

$$\langle \mathsf{let}\ x_1 = c_1[0]\ \mathsf{in}\ g_1, \mathsf{let}\ x_2 = c_2[0]\ \mathsf{in}\ g_2 \rangle$$

where $c_i[0]$ denotes the singleton element inhabiting $\{v \in \mathsf{v}(g_i) | p_i(v)\}|$.
With this syntactic sugar we have a 1-line specification for the complete graph.
$\mathsf{complete}(0) = 0$

$\mathsf{complete}(n) = \langle \mathsf{for}(x \leftarrow \mathsf{v}(\mathsf{discrete}(1)))\{\mathsf{discrete}(1)\}, \mathsf{for}(y \leftarrow \mathsf{v}(\mathsf{complete}(n-1)))\{\mathsf{complete}(n-1)\}\rangle$

These examples illustrate our main motivation for introducing the theory of graphs. As we move from discrete to chain to cycle to complete graphs the complexity of the graph structure grows dramatically; and yet the complexity of the recursive specifications of the graphs remains almost constant.

It is also important to recognize that each of these examples are also subdomains of their respective graph domains. For example,

$$\{\mathsf{chain(i)} | i \in \mathsf{Nat}\} \subset \mathsf{G}[\mathsf{Nat}, \mathsf{Nat}]$$

In this sense we can think of $\mathsf{G}[\mathsf{Nat}, \mathsf{Nat}]$ as a container domain for $\{\mathsf{chain(i)} | i \in \mathsf{Nat}\}$. More standard language would dub the latter a subtype of the former. In each of these examples we arrive at the subdomain or subtype of the container domain by imposing additional constraints on edge formation. In fact, each of these examples and infinitely many more can be computed either directly as we have done or as filters on their container domains, or super types, that selects graphs that adhere to the edge criteria or they can be computed by adjoining to the theory the additional constraints on edges. We will formalize this idea in a subsequent section.

The cycle and complete graph examples raise an important point about graph equality and references to whole graphs. In the section below we develop a notion of graph references to allow for more sophisticated compositional and recursive specification of graphs.

Graph references In many instances it it useful to refer not only to vertices but whole graphs. For this we assume that the collection of variables is actually divided into two distinct sub-collections; that is, we require $X = X_v + X_g$. Notationally, we will use lowercase letters such as x, y, z to range over X_v and uppercase letters such as A, B, C, etc to range over X_g. When we are building graphs using references to graphs we need to expand the theory to include more common cyclic definitions as is typically found in functional languages involving letrec.

Just as we separate the collection of variables into variables for vertices and variables for graphs, we separate the functional dependencies. This requires expanding the basic judgment to include graph variable dependencies on the left hand side of the turnstile: $\mathsf{G}[X, V]; \Gamma; \mathcal{G} \vdash g$. This expanded form allows mentions of A, B, etc to occur in g, which can be discharged later and gives rise to our first new rule. Just as with Γ, we use $\mathcal{G}$ to range over comma separated sequences of graph variables. This allows us to form wires.

$$\frac{}{\mathsf{G}[X, V]; \Gamma; A \vdash A} Wire$$

Equipped with this rule we can form graphs that have "holes" in them. Note that we could have achieved the same effect by calculating a graph context type from our graph type. However, with this next rule we allow bundling of wires and establish that our contexts have multiple holes.

$$\frac{\mathsf{G}[X,V];\Gamma_1;\mathcal{G}_1 \vdash g_1 \ \mathsf{G}[X,V];\Gamma_2;\mathcal{G}_2 \vdash g_2}{\mathsf{G}[X,V];\mathcal{G}_1,\mathcal{G}_2 \vdash g_1 \otimes g_2} Bundle[\Gamma_1 \cap \Gamma_2 = \emptyset = \mathcal{G}_1 \cap \mathcal{G}_2]$$

We can define a cut rule for graph variables just as we defined a cut rule for vertex variables.

$$\frac{\mathsf{G}[X,V];\Gamma_1;\mathcal{G}_1 \vdash g_1 \ \ \mathsf{G}[X,V];\Gamma_2;\mathcal{G}_2,B \vdash g_2}{\mathsf{G}[X,V];\boldsymbol{A},\boldsymbol{C} \vdash \mathsf{let}\ B = g_1\ \mathsf{in}\ g_2} Cut$$

This in turn makes it possible to describe a wide range of graphs with recursive structure. In particular, prior to the addition of graph references the theory describes only *finite* graphs built from X and V. With the addition of graph references the graphs can be *infinitary.* In the sequel we will refer to graphs that make no mention of graph variables as *ground.* Likewise, if graph g enjoys wellformedness without any dependencies, i.e. $\mathsf{G}[X,V];();() \vdash g$, we say g is *closed.*

Rewrite systems

The λ-calculus Let M denote the set of terms in the λ-calculus, and t, u, ... range over terms in M. We form graphs of the reductions of terms, $\mathsf{G}[M,M]$ by filtering edges for β-reduction. To avoid confusion in the sequel we will write $\ulcorner t \urcorner$ when using t as a variable, and unadorned when considering it as a term. More concretely, given

$$\uparrow t = \mathsf{let}\ \ulcorner t \urcorner = t\ \mathsf{in}\ [\ulcorner t \urcorner]$$

we lift terms to graphs using

$$\frac{}{\mathsf{G}[M,M];();() \vdash \uparrow t} Embedding$$

and then filter edges by the condition

$$\frac{\mathsf{G}[M,M];();() \vdash \uparrow t \ \ \mathsf{G}[M,M];();() \vdash \uparrow u \ \ t \to u}{\mathsf{G}[X,V];();() \vdash \langle \uparrow t, \uparrow u \rangle} Reduction$$

This gives rise to the graph of reductions of a term, $\mathsf{red}(t)$ which is defined by
$\mathsf{red}(x) = \uparrow x$
$\mathsf{red}(\lambda x.t) = \uparrow (\lambda x.t)$
$\mathsf{red}((\lambda x.t)u) = \langle \uparrow ((\lambda x.t)u), \mathsf{red}(t[u/x]) \rangle$
$\mathsf{red}(tu) = \langle \uparrow (tu), \mathsf{for}(t' \leftarrow \mathsf{v}(\mathsf{red}(t))\{\mathsf{red}(t'u)\} \rangle$

Categories We can provide a theory of categories, $\mathcal{C}[X,V]$, that are subdomains of $\mathsf{G}[X,V]$.

Reflection and the syntax of a graph In some sense the entire point of the previous sections is to provide proof that the domain $\mathsf{G}[X, V]$ serves equally well as theory of variables or as a theory of vertices, provided that X and V serve those roles respectively. That is, $\mathsf{G}[X, V]$ comes with an effective procedure for deciding when a graph, say g is in the domain, $\mathsf{G}[X, V]$. Specifically, we define $g \in \mathsf{G}[X, V]$ to be the determination that $\mathsf{G}[X, V]; (); () \vdash g$, i.e. that g is closed in $\mathsf{G}[X, V]$. Likewise, $\mathsf{G}[X, V]$ comes with an effective procedure for determining if two graphs are equal. This observation yields additional expressive power. We can consider graph domains that are recursively defined.

$$\mathcal{D}[X, V] = \mathsf{G}[X + \mathcal{D}[X, V], V + \mathcal{D}[X, V]]$$

Domains like $\mathcal{D}[X, V]$ allow us to create graphs in which we freely mix vertices from V with vertices made out of entire graphs. As an example, with the recursive domain we given a graph $g \in \mathsf{G}[X, V]$ we can compute the graph of its syntax, $\mathsf{s}(g) \in \mathcal{D}[X, V]$.

$\mathsf{s}(0) = \ulcorner 0 \urcorner | 0$
$\mathsf{s}(v|g) = (\mathsf{let}\ \ulcorner g \urcorner = g\ \mathsf{in}\ \ulcorner g \urcorner | \mathsf{s}(g), \mathsf{let}\ \ulcorner v \urcorner = v\ \mathsf{in}\ \ulcorner v \urcorner | 0)$
$\mathsf{s}(g_1 \otimes g_2)$
$=$
$(\mathsf{let}\ \ulcorner g_1 \urcorner = g_1\ \mathsf{in}\ \ulcorner g_1 \urcorner | \mathsf{s}(g_1), \mathsf{let}\ \ulcorner g_2 \urcorner = g_2\ \mathsf{in}\ \ulcorner g_2 \urcorner | \mathsf{s}(g_2))$
$\mathsf{s}(\mathsf{let}\ x = v\ \mathsf{in}\ g)$
$=$
$(\mathsf{let}\ \ulcorner g \urcorner = g\ \mathsf{in}\ \ulcorner g \urcorner | \mathsf{s}(g), \mathsf{let}\ \ulcorner x \urcorner = x\ \mathsf{in}\ \mathsf{let}\ \ulcorner v \urcorner = v\ \mathsf{in}\ \ulcorner x \urcorner | \ulcorner v \urcorner | 0)$
$\mathsf{s}((\mathsf{let}\ x_1 = v_1\ \mathsf{in}\ g_1, \mathsf{let}\ x_2 = v_2\ \mathsf{in}\ g_2))$
$=$
$(\mathsf{let}\ \ulcorner \mathsf{let}\ x_1 = v_1\ \mathsf{in}\ g_1 \urcorner = \mathsf{let}\ x_1 = v_1\ \mathsf{in}\ g_1\ \mathsf{in}\ \ulcorner \mathsf{let}\ x_1 = v_1\ \mathsf{in}\ g_1 \urcorner | \mathsf{s}(g_1),$
$\mathsf{let}\ \ulcorner \mathsf{let}\ x_2 = v_2\ \mathsf{in}\ g_2 \urcorner = \mathsf{let}\ x_2 = v_2\ \mathsf{in}\ g_2\ \mathsf{in}\ \ulcorner \mathsf{let}\ x_2 = v_2\ \mathsf{in}\ g_2 \urcorner | \mathsf{s}(g_2))$

It is easy to see that $\mathcal{S}[\mathcal{X}, \mathcal{V}] = \{\mathsf{s}(\})|\} \in \mathcal{D}[\mathcal{X}, \mathcal{V}]\}$ is a subtype of $\mathcal{D}[X, V]$.

And given the syntax of a graph we can always recover the original graph.

1.2 OSLF and Graphs

We can apply the OSLF procedure to the theory of graphs. When the collection is a set then the types are given as

$$\phi, \psi ::= \mathsf{true} \mid \phi\ \mathsf{and}\ \psi \mid 0 \mid v|\phi \mid \mathsf{let}\ x = v\ \mathsf{in}\ \phi \mid \langle \mathsf{let}\ x_1 = v_1\ \mathsf{in}\ \phi, \mathsf{let}\ x_2 = v_2\ \mathsf{in}\ \psi \rangle$$

This sets up our foundational investigation. Recall that our examples $\mathsf{discrete}$, chain, cycle, and $\mathsf{complete}$ gave rise to subtypes of $\mathsf{G}[\mathsf{Nat}, \mathsf{Nat}]$ by exploiting structure of Nat. Similarly, the OSLF types give rise to subtypes of $\mathsf{G}[X, V]$. Our focus of investigation is the question of the conditions under which the structure of X and V supply subtypes of $\mathsf{G}[X, V]$ that coincide with the OSLF types?

Not surprisingly, we find that when we can faithfully map X (resp., V) into $\mathsf{G}[X', V']$ for some X' and V', then we can establish an isomorphism between the subtypes available from the structure of X and V and the OSLF types. This observation sets up a process by which domains of X and V are replaced by graph-based representations of them. The limit of this process, the smallest structure for which these coincide, is given by

$$\mathcal{D} = \mathsf{G}[1 + \mathcal{D}, 1 + \mathcal{D}]$$

This domain is the least witness of the microcosm principle applied to the domain of graphs.

1.3 Miscellaneous operations

$$\mathsf{G}^{\circ}[X, V] = \mathsf{G}[V, X]$$

$$\mathsf{G}^{\mathsf{V}}[X, V] = \mathsf{G}[V, V]$$

$$\mathsf{G}^{\mathsf{X}}[X, V] = \mathsf{G}[X, X]$$

$$\mathsf{G}[X_1, V_1] + \mathsf{G}[X_2, V_2] = \mathsf{G}[X_1 + X_2, V_1 + V_2]$$

$$\mathsf{G}[X_1, V_1] \times \mathsf{G}[X_2, V_2] = \mathsf{G}[X_1 \times X_2, V_1 \times V_2]$$

$$\mathsf{G}[X_1, V_1] \cdot \mathsf{G}[X_2, V_2] = \mathsf{G}[X_1 + V_1, X_2 + V_2]$$

Note that the crossover style operations like "·" together with reflection allow us to define *evolutionary* processes (in the sense of Holland [5]) for graphs and graph domains. The principal missing ingredient is a fitness function for evaluating graphs and graph domains for selection and "reproduction."

2 Conclusions and future work

We have presented a formal theory of graphs that enjoys good complexity properties by comparison to other formal models.

Acknowledgments. The author wishes to thank Phillipa Gardner, Giorgio Ghelli, and Luca Cardelli for their inspiring work which lead to these insights.

References

1. Atish Bagchi and Charles Wells. Graph-based logic and sketches. *CoRR*, abs/0809.3023, 2008.
2. Luca Cardelli, Philippa Gardner, and Giorgio Ghelli. A spatial logic for querying graphs. In *ICALP*, volume 2380 of *Lecture Notes in Computer Science*, pages 597–610. Springer, 2002.
3. Anuj Dawar, Philippa Gardner, and Giorgio Ghelli. Expressiveness and complexity of graph logic. *Inf. Comput.*, 205(3):263–310, 2007.
4. C. Godsil and G.F. Royle. *Algebraic Graph Theory.* Graduate Texts in Mathematics. Springer New York, 2013.
5. John H. Holland. *Adaptation in natural and artificial systems.* University of Michigan Press, Michigan, USA, 1975.