A Parameterized Graph Transformation Calculus for Finite Graphs with Monadic Branches

Kazuyuki Asada
The University of Tokyo

Soichiro Hidaka
National Institute of Informatics

Hiroyuki Kato
National Institute of Informatics

Zhenjiang Hu
National Institute of Informatics

Keisuke Nakano
The University of Electro-Communications

ABSTRACT

We introduce a lambda calculus λ^T_{FG} for transformations of *finite* graphs by generalizing and extending an existing calculus UnCAL. Whereas UnCAL can treat only unordered graphs, λ^T_{FG} can treat a variety of graph models: directed edge-labeled graphs whose branch styles are represented by monads T. For example, λ^T_{FG} can treat unordered graphs, ordered graphs, weighted graphs, probability graphs, and so on, by using the powerset monad, list monad, multiset monad, probability monad, respectively. In λ^T_{FG}, graphs are considered as extension of tree data structures, i.e. as infinite (regular) trees, so the semantics is given with bisimilarity.

A remarkable feature of UnCAL and λ^T_{FG} is structural recursion for graphs, which gives a systematic programming basis like that for trees. Despite the non-well-foundedness of graphs, by suitably restricting the structural recursion, UnCAL and λ^T_{FG} ensures that there is a termination property and that all transformations preserve the finiteness of the graphs. The structural recursion is defined in a "divide-and-aggregate" way; "aggregation" is done by connecting graphs with ε-edges, which are similar to the ε-transitions of automata. We give a suitable general definition of bisimilarity, taking account of ε-edges; then we show that the structural recursion is well defined with respect to the bisimilarity.

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Language Classifications—*Specialized application languages*; E.1 [**Data Structures**]: *Graphs and networks*

General Terms

Language, Theory

Keywords

graph transformation, bisimilarity, structural recursion, graph algebra, monad, epsilon edge

1. INTRODUCTION

Designing graph transformation languages is difficult in general, because graph structured data may have cycles and shared nodes. There are three major difficulties: consistency, finiteness-preservation, and termination of evaluation. The consistency indicates the same treatment of 'equivalent' graphs, where equivalence is usually specified by either bisimilarity or isomorphism. Graphs up to bisimilarity are equivalent to infinite trees, whereas graphs up to isomorphism are "graphs as they look". In this paper, we focus on graphs up to bisimilarity for the application to querying tree databases with cyclic references such as XML with IDREF. Ensuring consistency requires that every graph transformation is *bisimulation generic*: i.e., the transformation results of bisimilar graphs are bisimilar.

For finiteness-preservation, it means that every term maps finite graphs (graphs with finite number of nodes) to finite graphs; this and the termination properties are important for graph query languages. Full recursion, which is a typical way of constructing powerful transformations, is not desirable, since the termination and finiteness-preservation do not hold. Structural recursion (fold) for inductive datatypes such as lists and finite trees naturally has both properties; however for graphs, because of the non-well-founded nature, it is difficult to find such a structural recursion. Functions defined by corecursion (unfold) terminate, but do not necessarily return finite graphs.

One prominent step to solve these problems was given in the work of UnCAL [6]. Their structural recursion for graphs is suitably restricted from that for infinite trees so that the termination and finiteness-preservation hold. The expressive power is less than that of fold for finite trees, but it is still powerful enough to translate an SQL-style graph query language—called UnQL—into UnCAL. Moreover, they gave bulk semantics for the structural recursion, which is "divide-and-aggregate" style semantics and enables parallel evaluation. The key to defining the structural recursion of UnCAL is use of ε-edges, which work as shortcut like the ε-transitions of automata. Then the bulk semantics makes it easier to prove the finiteness-preservation and termination properties. UnCAL is used also as a basis for bidirectional graph transformation [15] and for model driven software engineering [27].

However, there is a big restriction on UnCAL: graph models of UnCAL are just unordered graphs, i.e., graphs whose branches have no sibling order; while in real applications, there are many graph models such as ordered graphs, weighted graphs, probability graphs, etc. Though, when we generalize the graph model of UnCAL, we come across a subtle problem: how can we generally define the notion of bisimilarity for each graph model that has the

notion of ε-*edge*?

Additionally, we should point out that UnCAL has another weak point. The structural recursion of UnCAL is a higher order function like the fold for trees, but the bisimulation genericity of the structural recursion in UnCAL has been proved only on the first order part, and UnCAL is formulated as just first order calculus, which blurs how to extend UnCAL to richer type systems.

In previous work [14], the authors studied how to modify the graph model of UnCAL so that we can treat ordered graphs in a similar way to UnCAL. There it is found that ε-edges of ordered graphs may produce a subtle problem of *infinite width* on branches; this problem does not occur in the case of unordered graphs. It was resolved by proposing a new calculus for finite ordered graphs, giving a new definition of bisimilarity for ordered graphs having ε-edges, and giving the semantics of the calculus for ordered graphs. The calculus is extended with higher order functions and formulated as an extension of simply typed lambda calculus.

Along that line, in this paper we further generalize the graph models, and propose a family of calculi λ^T_{FG}, which is parametrized by monads T; each λ^T_{FG} can treat graphs whose branches are represented by a monad T. For example, when T is the finite powerset monad, λ^T_{FG} becomes UnCAL, precisely, UnCAL extended as a simply typed lambda calculus. To introduce λ^T_{FG}, we generalize all the features of UnCAL with monads: i.e., its graph models, its graph constructors, and the structural recursion.

In Section 2, we generalize the graph models from unordered graphs to T-graphs, by using a monad T (with some reasonable assumption) to represent various branch styles. When T is the list monad, the finite multiset monad, and the finite probability distribution monad, then T-graphs are ordered graphs, weighted graphs, and probability graphs, respectively. We also give a general definition of bisimilarity for T-graphs, taking ε-edges into account.

In Section 3, we introduce the syntax and the semantics of the lambda calculi λ^T_{FG}. We give the definition of graph constructors, and show that all finite T-graphs can be represented by those graph constructors. Then, we give a simple definition of the bulk semantics of the structural recursion for λ^T_{FG}, and we prove that all transformations of λ^T_{FG} are bisimulation generic. As well as generalizing with monads, λ^T_{FG} is extended from UnCAL with higher order functions, which is not for free but due to that our main theorem—bisimulation genericity of the bulk semantics—is stronger than the corresponding theorem for UnCAL [6].

In Section 4 we discuss related work, which includes more detail comparison between λ^T_{FG} and the calculus in [14].

Our contributions are summarized as follows:

- We generalize the graph model of UnCAL with monads by applying coalgebra theory; also we present a general definition of semantic equivalence for graphs having ε-edges, which is given by combination of ordinary bisimilarity and ε-*elimination*.
- We generally define graph constructors and structural recursion for T-graphs, with identifying suitable assumptions on the finiteness of T for the sake of practical use, which is discussed in Sections 2.2.1 and 3.2.4.
- We show that any finite T-graph can be constructed using graph constructors; also we find and utilize several equational properties for graph constructors.
- We extend semantic equivalence for graphs to higher order functions and show the bisimulation genericity of structural recursion as a higher order function, which enables us to reformulate UnCAL to more powerful and familiar style of calculus, i.e., the simply typed lambda calculus λ^T_{FG}.
- The reformulation makes it clear that we can further extend λ^T_{FG} with familiar features such as coproduct, algebraic datatypes, polymorphic types, dependent types, and so on. In fact, although the current formulation of λ^T_{FG} (and also UnCAL) does not include transformations for manipulating the sibling direction—e.g., reversing the order of branches of an ordered graph—we can add such transformations, using the above flexibility of extending λ^T_{FG}. (See [2] for the details of such an extension for sibling transformations.)

2. GRAPH MODEL AND BISIMILARITY

First, we explain what kind of graphs the λ^T_{FG}-terms can transform. After that, we give the semantic equivalence of the graphs: i.e., bisimilarity with ε-*elimination*.

2.1 Graph Model of λ^T_{FG}

The graphs in λ^T_{FG} are rooted, directed, and edge-labeled graphs. Furthermore, the graph model of λ^T_{FG} has two notable features: ε-*edges* and *markers*. An ε-edge represents a shortcut between the two nodes; the shortcut works like the ε-transition in an automaton. Nodes may be marked with *input* and *output markers*; these are used as interfaces to connect a graph to another graph by ε-edges. (This is done by @ and $\mathbf{cycle}$ as the dotted edges in Figure 4, and by $\mathbf{srec}$ as in Figure 5.)

Let us introduce the notion of a T-graph, which has "T-kind of branches", for a monad T. First, though, let us recall the notion of a monad (in the Kleisli triple style): a *monad* on $\mathbf{Set}$ is a triple $T = (T, \mathit{return}, \mathit{lift})$ of functions

$$\begin{aligned} T &\colon |\mathbf{Set}| \to |\mathbf{Set}| \\ \mathit{return}_S &\colon S \to T(S) \quad (S \in |\mathbf{Set}|) \\ \mathit{lift}_{S,S'} &\colon (S \to T(S')) \to (T(S) \to T(S')) \quad (S, S' \in |\mathbf{Set}|) \end{aligned}$$

such that these satisfy certain axioms (see [23, 4] for the axioms).

Example 1 (List Monad). The list functor List forms a monad with the following monad structures:

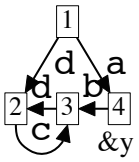
$$\begin{aligned} \mathit{return}(x) &= [x] \\ \mathit{lift}\ f\ xs &= \mathit{concat}\ (\mathit{map}\ f\ xs) \end{aligned}$$

where concat is to flatten a list of lists to a list. □

Now, let us define the graph model. We use $\mathcal{L}$ to denote a set of *labels* and $\mathcal{L}_\epsilon$ to denote the disjoint union $\mathcal{L} \cup \{\varepsilon\}$. Let X and Y be finite sets of *markers*; we add the prefix & for meta-variables of markers like $\&x$. Then, a T-*graph* (or just *graph*) G is defined as a triple (V, B, I) where

- V is a set of *nodes*,
- $B \colon V \to T(\mathcal{L}_\epsilon \times V + Y)$ is a *branch function*, where an element x in $\mathcal{L}_\epsilon \times V + Y$ (called *a branch*) is either *an edge* $\mathsf{Edge}\,(l, v)$ or *an output marker* $\mathsf{Outm}\,(\&y)$, and
- $I \colon X \to V$ is a function, which determines *input nodes* (*roots*) of the graph.

In terms of coalgebra theory, a T-graph is a coalgebra B of the endofunctor $T(\mathcal{L}_\epsilon \times (\text{-}) + Y)$ equipped with $I \colon X \to V$, which can be regarded as a generalized element (state) of V.



$$\begin{array}{lcl} V & = & \{1,2,3,4\} \\ B(1) & = & [\mathsf{Edge}\,(\mathtt{d},2)\,,\mathsf{Edge}\,(\mathtt{a},4)\,] \\ B(2) & = & [\mathsf{Edge}\,(\mathtt{c},3)\,] \\ B(3) & = & [\mathsf{Edge}\,(\mathtt{d},2)\,] \\ B(4) & = & [\mathsf{Edge}\,(\mathtt{b},3)\,,\mathsf{Outm}\,(\&y)\,] \\ I(\&) & = & 1 \end{array}$$

Figure 1: Example of Ordered Graph

Example 2 (Unordered/Ordered/Weighted/Probability Graphs). For the finite powerset monad $P_{\rm fin}$, $P_{\rm fin}$-graphs are *unordered graphs*, which are (equivalent to) the graph model of UnCAL.

$List$-graphs are *ordered graphs* [14], where the branches are ordered. An example of an ordered graph is shown in Figure 1.

The *finite multiset monad* (bag monad) is defined as $M_{\rm fin}$:

$$M_{\rm fin}(S) \stackrel{\rm def}{=} \{\phi\colon S{\to}\mathbb{N} \mid \phi^{-1}(\mathbb{N}{-}\{0\})\colon \text{finite}\}.$$

Branches of $M_{\rm fin}$-graphs have the bag semantics (rather than set semantics of $P_{\rm fin}$), i.e., multiplicity (called *weight*) of an identical branch is not ignored.

The *finite probability distribution monad* $D_{\rm fin}$ is defined as

$$D_{\rm fin}(S) \stackrel{\rm def}{=} \{\phi\colon S{\to}[0,1] \mid \phi^{-1}((0,1])\colon \text{finite},\ \Sigma_s\phi(s){=}1\}.$$

The monad structure is defined as below: for $s \in S$, $return(s)$ is the Dirac delta function $\delta_s\colon S \to [0,1]$:

$$\delta_s(s) \stackrel{\rm def}{=} 1 \qquad \delta_s(x) \stackrel{\rm def}{=} 0\ (x \neq s)$$

and for $f\colon S \to D_{\rm fin}(S')$,

$$\begin{array}{l} \mathit{lift}(f)\colon D_{\rm fin}(S) \to D_{\rm fin}(S') \\ (\phi\colon S \to [0,1]) \mapsto \left(\begin{array}{r} \mathit{lift}(f)(\phi)\colon S' \to [0,1] \\ s' \mapsto \sum_{s\in S} \big(\phi(s)\cdot f(s)(s')\big) \end{array} \right). \end{array}$$

$D_{\rm fin}$-graphs have probabilistic branches. □

The set[1] of graphs—with X and Y as sets of input and output markers, respectively—is denoted by T-$\mathcal{G}_Y^X$; here, T may be omitted as $\mathcal{G}_Y^X$ if it is clear from the context. We call a T-graph a *finite* T-graph when V is a finite set, and write T-$\mathcal{G}_{\rm f}{}_Y^X$ for the set of finite T-graphs (in [6], $P_{\rm fin}$-$\mathcal{G}_{\rm f}{}_Y^X$ is written as DB_Y^X).

We allow a graph to have multiple roots: a multi-rooted graph is to a forest as a single-rooted graph is to a tree. For single-rooted graphs, we often use a *default marker* & to indicate the root and use $\mathcal{G}_Y$ to denote $\mathcal{G}_Y^{\{\&\}}$.

Note that a node can have several output markers—if a monad T is non-deterministic as the above examples—and an output marker can be put on several nodes; while, a node can be pointed by several input markers, but an input marker points just one node.

2.2 Graph Equivalence in $\lambda_{\rm FG}^T$: Bisimilarity with ε-elimination

In the remaining part of this section, we give a semantic equivalence of graphs of $\lambda_{\rm FG}^T$. As discussed in the introduction, we regard the graphs in $\lambda_{\rm FG}^T$ as an extended tree data structure, i.e., as infinite trees, so we shall use bisimilarity semantics rather than equality or isomorphism semantics.

[1] Precisely, this is not a set but a proper class, but this matter is resolved in usual way in coalgebra theory, if T is ranked. In this paper, we consider only ranked monads.

The main difficulty with our definition of bisimilarity is the treatment of ε-edges. We first explain *strong bisimilarity*, in which we regard the label ε as a usual label such as those in $\mathcal{L}$. Then, we define *bisimilarity*, which "skips" ε-edges.

Note that our notion of bisimilarity for the invisible label ε is different from *weak bisimilarity* for the invisible label τ in the context of process algebra [21]. One purpose of our use of ε-edges is to postpone the calculations of the graph constructors, structural recursion, and so on, but weak bisimilarity is unsuitable for expected properties of such graph transformations: e.g., weak bisimilarity can not ensure the associativity of the graph constructor $\cup$ (defined in the next section).

Now, let us recall the notion of a bisimulation relation for any endofunctor F on $\mathbf{Set}$. First, we define *relational lifting* $\widetilde{F}$ *of* F. For a relation $R \subseteq V{\times}V'$, i.e., for an inclusion $\langle r, r'\rangle\colon R \hookrightarrow V{\times}V'$, we obtain $\langle F(r), F(r')\rangle : F(R) \to F(V){\times}F(V')$; then the relation $\widetilde{F}(R) \subseteq F(V){\times}F(V')$ can be defined as the image $\langle F(r), F(r')\rangle(F(R))$. Next, for two *coalgebras* of F, i.e., two functions $B\colon V \to F(V)$ and $B'\colon V' \to F(V')$, a *bisimulation relation* R between B and B' is a relation $R \subseteq V{\times}V'$ such that $(B{\times}B')(R) \subseteq \widetilde{F}(R)$.

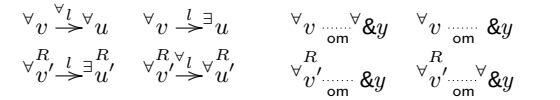
Definition 3 (Strong Bisimilarity). Let T be a monad, and $G = (V, B, I)$ and $G' = (V', B', I')$ be T-graphs in T-$\mathcal{G}_Y^X$. Then G and G' are *strongly bisimilar* if there is a bisimulation relation R with respect to the endofunctor $T(\mathcal{L}_\epsilon{\times}(\text{-})+Y)$ between B and B' such that for any $\&x \in X$, $I(\&x)\ R\ I'(\&x)$; in this case, we write $G \sim_{\rm s} G'$. □

We assume that all monads T in the paper preserve weak-pullbacks, which is a mild assumption often used in coalgebra theory [26, 30]. In particular, then T preserves injections and finite intersections. Using this assumption, it is easily checked that the strong bisimilarity relation is an equivalence relation on T-$\mathcal{G}_Y^X$.

Let us unfold the above abstract definition of strong bisimilarity, when $T = P_{\rm fin}$:

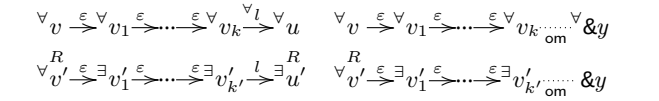
$$\begin{array}{l} \quad R\text{: a bisimulation relation for } P_{\rm fin}(\mathcal{L}_\epsilon{\times}(\text{-})+Y) \\ \Longleftrightarrow \forall (v,v'){\in}R.\ (B(v), B'(v')) \in \\ \qquad \langle P_{\rm fin}(\mathcal{L}_\epsilon{\times}r{+}Y), P_{\rm fin}(\mathcal{L}_\epsilon{\times}r'{+}Y)\rangle(P_{\rm fin}(\mathcal{L}_\epsilon{\times}R{+}Y)) \\ \Longleftrightarrow \forall (v,v'){\in}R.\ \exists S \subseteq_{\mathit{fin}} \mathcal{L}_\epsilon{\times}R{+}Y. \\ \qquad B(v){=}(\mathcal{L}_\epsilon{\times}r{+}Y)(S) \wedge B'(v'){=}(\mathcal{L}_\epsilon{\times}r'{+}Y)(S) \\ \Longleftrightarrow \forall (v,v'){\in}R.\ \exists \{(l_1,(u_1,u_1')),...,(l_n,(u_n,u_n')),\&y_1,...,\&y_m\} \\ \qquad \subseteq_{\mathit{fin}} \mathcal{L}_\epsilon{\times}R{+}Y. \\ \qquad B(v){=}\{(l_1,u_1),...,(l_n,u_n),\&y_1,...,\&y_m\} \wedge \\ \qquad B'(v'){=}\{(l_1,u_1'),...,(l_n,u_n'),\&y_1,...,\&y_m\} \\ \Longleftrightarrow \forall (v,v'){\in}R. \\ \qquad \big(\forall (l,u){\in}B(v).\ \exists u'.\ (l,u'){\in}B'(v') \wedge (u,u'){\in}R\big) \wedge \\ \qquad \big(\forall \&y{\in}B(v).\ \&y{\in}B'(v')\big) \wedge \\ \qquad \big(\forall (l,u'){\in}B'(v').\ \exists u.\ (l,u){\in}B(v){\wedge}(u,u'){\in}R\big) \wedge \\ \qquad \big(\forall \&y{\in}B'(v').\ \&y{\in}B(v)\big) \end{array}$$

The last formula can be expressed as in the following pictures.



Now, let us define our bisimilarity, whose instantiation to the case when $T = P_{\rm fin}$ should agree with the bisimilarity of

UnCAL [6]. The bisimilarity of UnCAL is defined by replacing the first and the third pictures above with the following two pictures (and similarly for the second and the fourth pictures).



With this bisimulation, we "skip" zero or more occurrences of ε-edges before a non-ε edge or an output marker. Categorically, this kind of iteration is captured by the notion of an *iteration operator*, which is the dual notion of a fixed-point operator [29, 17]. From now we define *ε-elimination*—which is almost the same as that for an ε-automaton—in terms of an iteration operator in the Kleisli category of a monad T, and then give a definition of bisimilarity for T-graphs.

For a monad T on $\mathbf{Set}$, the *Kleisli category* $\mathbf{Set}_T$ of T is defined as below: objects in $\mathbf{Set}_T$ are sets, and morphisms $S \to S'$ are functions $S \to T(S')$. The identity morphism $\underline{id}$ on S is

$$\underline{id} \stackrel{\text{def}}{=} return\colon S \to T(S),$$

and the composition $g \odot f$ of $f\colon S \to T(S')$ and $g\colon S' \to T(S'')$ is defined as

$$g \odot f \stackrel{\text{def}}{=} lift(g) \circ f\colon S \to T(S'').$$

A Kleisli category has coproducts: the coproduct of S_1 and S_2 is just $S_1{+}S_2$, and the injections are

$$\underline{in_l} \stackrel{\text{def}}{=} return \circ in_l\colon S_1 \to T(S_1{+}S_2)$$
$$\underline{in_r} \stackrel{\text{def}}{=} return \circ in_r\colon S_2 \to T(S_1{+}S_2).$$

Copairing of a pair of functions $f_1\colon S_1 \to T(S')$ and $f_2\colon S_2 \to T(S')$ is the same as the copairing in $\mathbf{Set}$, i.e.,

$$[f_1, f_2]\colon S_1{+}S_2 \to T(S').$$

We write $\underline{\nabla}\colon S{+}S \to T(S)$ and $\underline{+}$ for the codiagonal and the coproduct on morphisms in $\mathbf{Set}_T$, respectively.

Next we recall the notion of an iteration operator [10, 17]. Though we can define an iteration operator for any category with finite coproducts, here, we define it directly on the Kleisli category $\mathbf{Set}_T$ of a monad T on $\mathbf{Set}$, and say that *a monad T has an iteration operator* if $\mathbf{Set}_T$ has it. An *iteration operator* $iter$ on $\mathbf{Set}_T$ is an operator on functions

$$\frac{f\colon S \to T(S{+}A)}{iter(f)\colon S \to T(A)}$$

such that the operator satisfies the following axioms:

- (naturality:) for $f\colon S \to T(S{+}A)$ and $g\colon A \to T(A')$,
$$g \odot iter(f) = iter((\underline{id}_S \underline{+} g) \odot f)\colon S \to T(A'),$$
- (dinaturality:) for $f\colon S \to T(S'{+}A)$ and $g\colon S' \to T(S{+}A)$,
$$[iter([f, \underline{in_r}] \odot g), \underline{id}_A] \odot f = iter([g, \underline{in_r}] \odot f)\colon S \to T(A),$$
- (unfolding:) for $f\colon S \to T(S{+}A)$,
$$iter f = [iter f, \underline{id}_A] \odot f\colon S \to T(A),$$
- (codiagonal:) for $f\colon S \to T(S{+}S{+}A)$,
$$iter(iter f) = iter((\underline{\nabla} \underline{+} \underline{id}_A) \odot f)\colon S \to T(A).$$

Further, for a class $\mathcal{M}$ of morphisms of $\mathbf{Set}_T$, $iter$ is called *uniform on $\mathcal{M}$* if for any function $f\colon S \to T(S')$ in $\mathcal{M}$, and any functions $g\colon S' \to T(S'{+}A)$ and $h\colon S \to T(S{+}A)$,

$$iter(g) \odot f = iter(h)\colon S \to T(A)$$

whenever

$$g \odot f = (f \underline{+} \underline{id}_A) \odot h\colon S \to T(S'{+}A).$$

The axiom of uniformity is used for logical relation on an iteration operator [12]. We use uniformity to show later that strong bisimilarity implies bisimilarity, and also in the proof of Lemma 16.

Example 4 (Monads with Iteration Operators). For the countable powerset monad P_{cnt} and a morphism $f\colon S \to P_{\mathrm{cnt}}(S{+}S')$ in $\mathbf{Set}_{P_{\mathrm{cnt}}}$,

$$iter(f)(s) \stackrel{\text{def}}{=} \{s' \in S' \mid \exists k \in \mathbb{N}.\ \exists s_1, ..., s_k \in S.\ s_1 \in f(s) \wedge ... \wedge s_k \in f(s_{k-1}) \wedge s' \in f(s_k)\}.$$

Also, the *countable multiset monad* M_{cnt}:

$$M_{\mathrm{cnt}}(S) \stackrel{\text{def}}{=} \{\phi\colon S \to \mathbb{N} \cup \{\infty\} \mid \phi^{-1}(\mathbb{N} - \{0\}) \text{ is countable}\}$$

has an iteration operator, which is given with the same formula as that for P_{cnt}.

Extending the list monad—$List(S) \stackrel{\text{def}}{=} \coprod_{n \in \mathbb{N}} S^n$—, a *countable list monad* $CList$ is defined as

$$CList(S) \stackrel{\text{def}}{=} \coprod_{L \in \mathbb{L}} S^L,$$

where $\mathbb{N}$ is generalized to $\mathbb{L}$, the set of countable linear ordered sets up to order isomorphism[2]. $\mathbf{Set}_{CList}$ also has an iteration operator (see [14, 2] for the details).

For probability graphs, *countable subprobability distribution monad* $SubD_{\mathrm{cnt}}$ has an iteration operator:

$$SubD_{\mathrm{cnt}}(S) \stackrel{\text{def}}{=} \{\phi\colon S \to [0,1] \mid \phi^{-1}((0,1]) \text{ is countable}, \Sigma_x \phi(x) \leq 1\}.$$

Note that here the summation of probabilities $\Sigma_x \phi(x)$ is not necessarily 1; this is because the probability $1 - \Sigma_x \phi(x)$ is reserved for the probability of the nontermination of the iteration operator. The definition of the iteration operator for $SubD_{\mathrm{cnt}}$ is also similar to those for P_{cnt} and M_{cnt}, see [16] for the details. □

Now, let us define ε-elimination. The following characterization of ε-elimination as an iteration operator is due to [13, 16].

Definition 5 (ε-elimination). Let T be a monad and $iter$ be an iteration operator in $\mathbf{Set}_T$. For a T-graph $G = (V, B, I) \in T\text{-}\mathcal{G}_Y^X$, its *$\varepsilon$-elimination* $\varepsilon\text{-elim}(G) \in T\text{-}\mathcal{G}_Y^X$ is (V, B', I) where

$$B' \stackrel{\text{def}}{=} embed \circ iter(iso \circ B),$$

$embed$ is the embedding $T(\mathcal{L}{\times}V{+}Y) \to T(\mathcal{L}_\epsilon{\times}V{+}Y)$, and iso is the composition of

$$T(\mathcal{L}_\epsilon{\times}V{+}Y) \cong T((\mathcal{L}{+}1){\times}V{+}Y) \cong T(V{+}(\mathcal{L}{\times}V{+}Y)).$$

Conversely, ε-elimination induces an iteration operator; let us consider a T-graph (V, B, I) in the case that $\mathcal{L} = 0$. Then $B\colon V \to T(\{\varepsilon\}{\times}V{+}Y)$, and if we apply ε-elimination to this, the resulting

[2] More precisely, $CList(S)$ is the set of objects of the skeleton of the comma category $(U \downarrow S)$ where $U : \mathbf{CLO} \to \mathbf{Set}$ is the forgetful functor from the category $\mathbf{CLO}$ of countable linear ordered sets and monotone functions.

branch function is $B'\colon V \to T(0\times V+Y)$. That is, we get an operator that maps a function $V \to T(V+Y)$ to a function $V \to T(Y)$. This operator is the same as the structure of an iteration operator in the Kleisli category (if we allow Y to be arbitrary sets); and then we find that it is natural to adopt the axioms of iteration operators as axioms of the ε-elimination.

Now, let us define our bisimilarity for graphs having ε-edges.

Definition 6 (Bisimilarity). Let T be a monad having an iteration operator $iter$ in the Kleisli category $\mathbf{Set}_T$, and $G = (V, B, I)$ and $G' = (V', B', I')$ be T-graphs in $T\text{-}\mathcal{G}_Y^X$. Then, G and G' are *bisimilar* if $\varepsilon\text{-elim}(G)$ and $\varepsilon\text{-elim}(G')$ are strongly bisimilar; in this case, we write $G \sim G'$. □

It immediately follows that strong bisimilarity implies bisimilarity, if the iteration operator of T is uniform on the class of functions (rather than not necessarily on all morphisms in $\mathbf{Set}_T$). We use this property in some of the proofs in this paper.

When $T = P_{\mathrm{cnt}}$, by unfolding the above definition, we get the original definition of bisimilarity in UnCAL.

The above definition implies that the notion of an ε-edge a la ε-elimination is independent of bisimilarity semantics, and it can be accommodated in any equivalence relation, such as equality and graph isomorphism, by similarly taking inverse images.

2.2.1 Generalization of Bisimilarity with Monad Extension

Although we could define bisimilarity as above, when T is one of the monads in Example 4, the T-graphs might have countably infinite width of branches. In a real database system, a graph (is often very large but) has finite size, i.e., finite number of nodes and a finite width. In other words, real data graphs do not have ε-edges, which cause graphs to have infinite widths. Our use of ε-edge is just in an implementation of λ_{FG}^T for efficiency and for defining structural recursion; so we shall suppose that what users of λ_{FG}^T can observe is just finite-width graphs, and graphs whose ε-elimination have infinite widths are regarded as errors. (For any finite $List$-graph G, it is decidable whether the ε-elimination of G keeps finite width or not; see [14, 2].)

Therefore, the finite powerset monad and the other monads in Example 2 are more suitable for practical purposes; however, they do not have iteration operators. Hence, we do not require that T itself has an iteration operator in the Kleisli category, and instead we use the following assumption: we say that T *has an extension* T' *for* ε*-elimination* if we have a monad T', an injective monad morphism $\iota : T \hookrightarrow T'$, and an iteration operator in the Kleisli category of T' that satisfies the uniformity on the class of functions. Here, *monad morphism* is a natural transformation that is compatible with $return$'s and with $lift$'s (see [4] for the details).

Definition 7 (Bisimilarity Generalized on Size). Let T be a monad which has an extension T' for ε-elimination. Then there is an embedding $\iota\text{-}\mathcal{G}_Y^X\colon T\text{-}\mathcal{G}_Y^X \hookrightarrow T'\text{-}\mathcal{G}_Y^X$ which maps (V, B, I) to $(V, \iota_{(\mathcal{L}_\epsilon\times V+Y)}\circ B, I)$. For G and G' in $T\text{-}\mathcal{G}_Y^X$, G *and* G' *are bisimilar* if $\iota\text{-}\mathcal{G}_Y^X(G)$ and $\iota\text{-}\mathcal{G}_Y^X(G')$ are bisimilar in the sense of Definition 6. □

From the assumption that ι has injective components and T preserves weak pullbacks, $\iota\text{-}\mathcal{G}_Y^X$ reflects strong bisimilarity [30, Theorem 4.3.6]; hence, strong bisimilarity and the above bisimilarity are equivalent for T-graphs having no ε-edges.

In the next section, we assume that a monad T is *finitary* and see the usefulness; after that, in Section 3.2.4 we will come back to more detail discussion why and how we need an extension of a monad for ε-elimination.

$$
\begin{array}{rll}
\text{Term } e ::= & x \mid \lambda x.e \mid ee \mid (e,e) \mid \pi^{\mathbf{l}}e \mid \pi^{\mathbf{r}}e & \{\text{ lambda terms }\}\\
& \mid \mathbf{if}\ e\ \mathbf{then}\ e\ \mathbf{else}\ e & \{\text{ conditional }\}\\
& \mid \mathbf{op}_s & \{\ T\text{-algebraic graph constructors } (s\in\Sigma)\ \}\\
& \mid \langle e : e\rangle \mid \langle \&y\rangle \mid \&x := e \mid () \mid e\oplus e \mid e\ @\ e \mid & \\
& \mid \mathbf{cycle}(e) & \{\text{ common graph constructors }\}\\
& \mid \mathbf{srec}(e)(e) & \{\text{ structural recursion application }\}\\
& \mid a \mid e = e & \{\text{ label } (a\in\mathcal{L}) \text{ and label equality }\}\\
\text{Type } \sigma ::= & \mathbf{Bool} \mid \mathbf{Label} \mid \mathbf{G}_Y^X & \{\text{ boolean, label, graph types }\}\\
& \mid \sigma\times\sigma \mid \sigma\to\sigma & \{\text{ product types and function types }\}
\end{array}
$$

Figure 2: Syntax of λ_{FG}^T

$$
\frac{(s\in\Sigma, s\colon n\text{-arity})}{\Gamma\vdash \mathbf{op}_s : \mathbf{G}_Y^{X^n}\to\mathbf{G}_Y^X}
\qquad
\frac{\Gamma\vdash e_1\colon \mathbf{Label} \quad \Gamma\vdash e_2\colon \mathbf{G}_Y}{\Gamma\vdash \langle e_1\colon e_2\rangle : \mathbf{G}_Y}
$$

$$
\frac{(\&y\in Y)}{\Gamma\vdash\langle\&y\rangle\colon \mathbf{G}_Y}
\qquad
\frac{\Gamma\vdash e\colon \mathbf{G}_Y}{\Gamma\vdash \&x := e\colon \mathbf{G}_Y^{\{\&x\}}}
\qquad
\frac{}{\Gamma\vdash ()\colon \mathbf{G}_Y^{\emptyset}}
$$

$$
\frac{\Gamma\vdash e_1\colon \mathbf{G}_Y^{X_1} \quad \Gamma\vdash e_2\colon \mathbf{G}_Y^{X_2} \quad (X_1\cap X_2=\emptyset)}{\Gamma\vdash e_1\oplus e_2\colon \mathbf{G}_Y^{X_1\cup X_2}}
\qquad
\frac{\Gamma\vdash e_1\colon \mathbf{G}_Y^{X} \quad \Gamma\vdash e_2\colon \mathbf{G}_Z^{Y}}{\Gamma\vdash e_1\ @\ e_2\colon \mathbf{G}_Z^{X}}
$$

$$
\frac{\Gamma\vdash e\colon \mathbf{G}_{X\cup Y}^X \quad (X\cap Y=\emptyset)}{\Gamma\vdash \mathbf{cycle}(e)\colon \mathbf{G}_Y^X}
$$

$$
\frac{\Gamma,\ l\colon\mathbf{Label},\ g\colon\mathbf{G}_Y\vdash e_1\colon \mathbf{G}_Z^Z \qquad \Gamma\vdash e_2\colon \mathbf{G}_Y^X}{\Gamma\vdash \mathbf{srec}(\lambda(l,g).e_1)(e_2)\colon \mathbf{G}_{Z\times Y}^{Z\times X}}
$$

(Just unfamiliar rules are listed. We use l and g as meta variables for variables of types $\mathbf{Label}$ and $\mathbf{G}_Y^X$, respectively.)

Figure 3: Typing Rules of λ_{FG}^T

3. SYNTAX AND SEMANTICS OF λ_{FG}^T

Here, we give the syntax of λ_{FG}^T and its semantics. The semantics of λ_{FG}^T has two steps. The first step is an interpretation to just in the (V, B, I) form, without considering bisimilarity. This is equality-based semantics rather than bisimilarity-based one. The next step is to give an interpretation up to the bisimilarity; to do so, we need to show *bisimulation genericity* of terms, i.e., well-definedness in terms of bisimilarity. Since structural recursion is a higher order function, before showing bisimulation genericity, we will extend the equivalence relation of bisimilarity to function types.

3.1 Syntax of λ_{FG}^T

The syntax of λ_{FG}^T is given in Figure 2, and the typing rule is given in Figure 3. The syntax of λ_{FG}^T in fact depends on not only a monad T but also its signature Σ, which will be explained in the next subsection. However, the expressive power of λ_{FG}^T is independent of a choice of the signatures Σ.

3.2 Graph Constructors

Here, we give an interpretation of the graph constructors in λ_{FG}^T.

The original UnCAL and $\lambda_{\mathrm{FG}}^{P_{\mathrm{fin}}}$ have the nine graph constructors in Figure 4, by which all finite graphs can be represented. (In the original UnCAL, the graph constructors $\langle a : \text{-}\rangle$ and $\langle \&y\rangle$ are written as $\{a : \text{-}\}$ and $\&y$, respectively.) Note that these constructors should

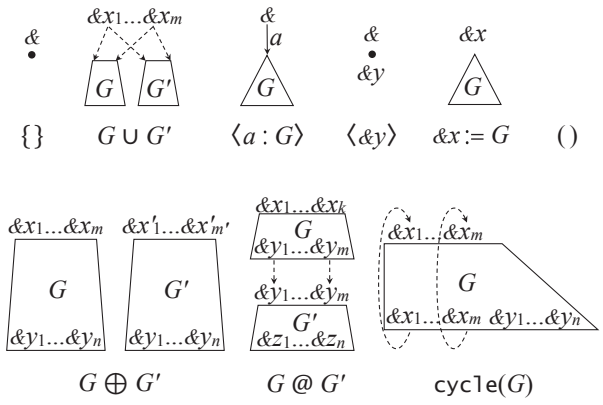


Figure 4: Nine Graph Constructors of $\lambda_{\mathrm{FG}}^{P_{\mathrm{fin}}}$

be written as $\{\}_Y$, $G_1 \cup_{X,Y} G_2$, and so on with type (marker-sets) annotation; however, we will omit the subscript X and Y to avoid clutter.

We separate the nine graph constructors into *common graph constructors* and *T-algebraic graph constructors*. When $T = P_{\mathrm{fin}}$, $\{\}$ and $\cup$ are T-algebraic graph constructors, and the other seven graph constructors are common graph constructors. Common graph constructors are defined independently of the difference of monads, while the definition of T-algebraic graph constructors depends on algebraic operations of T.

3.2.1 Common Graph Constructors

We first give a definition of common graph constructors.

Definition 8 (Common Graph Constructors). Let T be a monad.

- For $G = (V, B, I) \in \mathcal{G}_Y$,

$$\langle a : G\rangle \stackrel{\text{def}}{=} (V \cup \{v_0\colon \text{fresh}\},\ B',\ \{\&\mapsto v_0\}) \in \mathcal{G}_Y$$

  where $B'(v) \stackrel{\text{def}}{=} B(v)$ and $B'(v_0) \stackrel{\text{def}}{=} \mathit{return}\big(\mathsf{Edge}\,(a, I(\&))\big)$.

- For $\&y \in Y$,

$$\langle \&y\rangle \stackrel{\text{def}}{=} (\{\&\},\ \{\&\mapsto \mathit{return}(\mathsf{Outm}\,(\&y))\},\ \mathit{id}_{\{\&\}}) \in \mathcal{G}_Y.$$

- For $G = (V, B, I) \in \mathcal{G}_Y$,

$$(\&x := G) \stackrel{\text{def}}{=} (V, B, \{\&x \mapsto I(\&)\}) \in \mathcal{G}_Y^{\{\&x\}}.$$

- $() \stackrel{\text{def}}{=} (\emptyset,\ \text{“the unique function from } \emptyset\text{”},\ \mathit{id}_\emptyset) \in \mathcal{G}_Y^\emptyset$.

- For $G = (V, B, I) \in \mathcal{G}_Y^X$ and $G' = (V', B', I') \in \mathcal{G}_Y^{X'}$ such that $X \cap X' = \emptyset$,

$$G \oplus G' \stackrel{\text{def}}{=} (V{+}V',\ B'',\ I{+}I') \in \mathcal{G}_Y^{X\cup X'}$$

  where $B'' \stackrel{\text{def}}{=} [T(\mathcal{L}_\epsilon{\times}(\mathit{in}_l){+}Y) \circ B,\ T(\mathcal{L}_\epsilon{\times}(\mathit{in}_r){+}Y) \circ B']\colon\ V{+}V' \to T(\mathcal{L}_\epsilon{\times}(V{+}V'){+}Y)$.

- For $G = (V, B, I) \in \mathcal{G}_Y^X$ and $G' = (V', B', I') \in \mathcal{G}_Z^Y$,

$$G \mathbin{@} G' \stackrel{\text{def}}{=} (V{+}V',\ B'',\ \mathit{in}_l \circ I) \in \mathcal{G}_Z^X$$

  where

$$B''(\mathit{in}_l(v)) \stackrel{\text{def}}{=} T(f)(B(v))$$
$$\left(\begin{array}{l} f\colon \mathcal{L}_\epsilon{\times}V{+}Y \to \mathcal{L}_\epsilon{\times}(V{+}V'){+}Z \\ \quad \mathsf{Edge}\,(l, v) \mapsto \mathsf{Edge}\,(l, \mathit{in}_l(v)) \\ \quad \mathsf{Outm}\,(\&y) \mapsto \mathsf{Edge}\,\big(\varepsilon, \mathit{in}_r(I'(\&y))\big) \end{array}\right)$$
$$B''(\mathit{in}_r(v')) \stackrel{\text{def}}{=} (T(\mathcal{L}_\epsilon{\times}(\mathit{in}_r){+}Z))(B'(v'))$$

- For a graph $G = (V, B, I) \in \mathcal{G}_{X\cup Y}^X$ such that $X \cap Y = \emptyset$,

$$\mathbf{cycle}(G) \stackrel{\text{def}}{=} (V, B', I) \in \mathcal{G}_Y^X$$

  where $B'(v) \stackrel{\text{def}}{=} T(f)(B(v))$.

$$\left(\begin{array}{l} f\colon \mathcal{L}_\epsilon{\times}V{+}(X \cup Y) \to \mathcal{L}_\epsilon{\times}V{+}Y \\ \quad \mathsf{Edge}\,(l, v) \mapsto \mathsf{Edge}\,(l, v) \\ \quad \mathsf{Outm}\,(\&x) \mapsto \mathsf{Edge}\,(\varepsilon, I(\&x)) \\ \quad \mathsf{Outm}\,(\&y) \mapsto \mathsf{Outm}\,(\&y) \end{array}\right)$$

□

Remark for Figure 4: Recall that, for one graph G in $\mathcal{G}_Y^X$ and one marker $\&y \in Y$, there may be more than one or zero $\&y$-occurrences in G, while for one marker $\&x \in X$, there must be just one $\&x$-input node in G. Hence, for each $\&y_i$ in $G \mathbin{@} G'$ in the figure, although the number of ε-edges from a node with $\&y_i$-output marker seems just one, in fact we add the same number of ε-edges as the number of $\&y_i$-output marker occurrences in G. This is the same for $\mathbf{cycle}(G)$; but, on $G \cup G'$, the number of added ε-edges are just $2m$.

3.2.2 T-Algebraic Graph Constructors

Here, we will define the T-algebraic graph constructors. The definition depends on algebraic operations of a monad T, and the syntax for T-algebraic graph constructors depends on a signature Σ generating the monad T. For example, a finite powerset $P_{\mathrm{fin}}(X)$ is a free semilattice, which has two algebraic operators, i.e., bottom and join. In the syntax of $\lambda_{\mathrm{FG}}^{P_{\mathrm{fin}}}$, when we choose the signature Σ of (the function symbols of) bottom and join, then $\mathbf{op}_s$ ($s \in \Sigma$) have the same interpretation as the two graph constructors $\{\}$ and $\cup$ in the original UnCAL. Below, we first define the interpretation of T-algebraic graph constructors without considering signatures, and after that we consider signatures for syntax of λ_{FG}^T.

Later, we assume T to be finitary in order to prove Proposition 11. A *finitary monad* on $\mathbf{Set}$ is a monad T on $\mathbf{Set}$ such that the functor T preserves all directed colimits in $\mathbf{Set}$; in other words, for any set S and any $x \in T(S)$, there is a finite subset $S' \subseteq S$ such that $x \in T(S')$. The monads P_{fin}, List, M_{fin}, and D_{fin} are all finitary monads; for example, for $[4, 1, 6, 4, 6, 4] \in \mathit{List}(\mathbb{N})$, we have a finite subset $\{4, 1, 6\} \subseteq \mathbb{N}$ and $[4, 1, 6, 4, 6, 4] \in \mathit{List}(\{4, 1, 6\})$.

Here, though we define graphs in (V, B, I) form, when we consider the property of graph operations, we consider them up to bisimilarity. Hence, we assume that T has an extension for ε-elimination.

First, we explain that defining T-algebraic graph constructors for multi-rooted graphs is reduced to that for single-rooted graphs. Note that the multi-rootedness is semantically the same as the power of sets of graphs: i.e., there is the following bijection.

$$\begin{aligned} \mathit{oplus}\colon (\mathcal{G}_{\mathrm{f}\,Y})^X &\stackrel{\cong}{\to} \mathcal{G}_{\mathrm{f}\,Y}^X \\ f &\mapsto \bigoplus_{\&x\in X} \&x := f(\&x) \\ (\&x \mapsto \langle \&x\rangle \mathbin{@} G) &\leftarrow\!\shortmid G \end{aligned}$$

For an n-ary operator $o\colon S^n \to S$ on a set S, there is the obvious n-ary operator $o^{(X)}$ on the X-th power of S: i.e., the com-

position of $(S^X)^n \xrightarrow{\cong} (S^n)^X \xrightarrow{o^X} S^X$; we call this *power algebra*. Then, when $T = P_{\rm fin}$, i.e., for UnCAL, it can be shown that $\cup\colon (\mathcal{G}_{{\rm f}Y}^X)^2 \to \mathcal{G}_{{\rm f}Y}^X$ is nothing but the X-th power algebra of $\cup\colon (\mathcal{G}_{{\rm f}Y})^2 \to \mathcal{G}_{{\rm f}Y}$. Hence, we will define T-algebraic graph constructors with the type $(\mathcal{G}_{{\rm f}Y})^n \to \mathcal{G}_{{\rm f}Y}$.

Now, for a finitary monad T, let us define a function $T^{\mathcal{G}}$ as $T^{\mathcal{G}}(X) \stackrel{\rm def}{=} T\text{-}\mathcal{G}_{{\rm f}X}$ for a finite set X. As we will soon show, $T^{\mathcal{G}}$ itself can be extended to a finitary monad. Then, for constructing operators of the type $(\mathcal{G}_{{\rm f}Y})^n \to \mathcal{G}_{{\rm f}Y}$, i.e., $T^{\mathcal{G}}(Y)^n \to T^{\mathcal{G}}(Y)$ we can use the result [24] that an element s in $T(n)$ bijectively corresponds to a family $o(s)$ of n-ary operators $(o(s)_Y\colon T(Y)^n \to T(Y))_{Y\colon \rm set}$ that is natural on $Y \in \mathbf{Set}_T$. In this context, s is called a *generic effect*, and $o(s)$ is called an *algebraic operation*. For $s \in T(n)$, $o(s)$ is defined as

$$o(s)_Y\colon T(Y)^n (= n \to T(Y)) \to T(Y) \qquad (1)$$
$$f \mapsto \mathit{lift}^T(f)(s)$$

and for $(o_Y\colon T(Y)^n \to T(Y))_{Y\colon \mathbf{Set}_T}$, the corresponding s is defined as $o_n(\mathit{return}_n^T)$. For example, when $T = P_{\rm fin}$, $\{0,1\} \in P_{\rm fin}(2)$ ($2 = \{0,1\}$) corresponds to $\cup\colon P_{\rm fin}(Y)^2 \to P_{\rm fin}(Y)$. From now, we apply this correspondence to the case of $T^{\mathcal{G}}$ as T. Then we will show that, for a given finitary monad T, there is an embedding of T into $T^{\mathcal{G}}$, so that we can define T-algebraic graph constructors by algebraic operations of T.

Now, let us define the monad $T^{\mathcal{G}}$. Here, we consider only finite sets, since it is enough here (and also since finitary monads are determined by definitions for finite sets by the left Kan-extension as in [18, Proposition 7.6]). The monad structures of $T^{\mathcal{G}}$ are defined as follows: $\mathit{return}^{T^{\mathcal{G}}}\colon X \to \mathcal{G}_{{\rm f}X}$ maps $\&x$ to $\langle \&x \rangle$, and $\mathit{lift}^{T^{\mathcal{G}}}\colon (X \to \mathcal{G}_{{\rm f}Y}) \to (\mathcal{G}_{{\rm f}X} \to \mathcal{G}_{{\rm f}Y})$ maps f to $(\text{-}) \mathbin{@} \mathit{oplus}(f)$. Thus, a graph in $\mathcal{G}_{{\rm f}Y}^X (\cong (\mathcal{G}_{{\rm f}Y})^X)$ can be regarded as a "function" from X to Y (precisely, a morphism in the Kleisli category of $T^{\mathcal{G}}$); then the composition is given by @ operator. For later use, for $f\colon X \to Y$, we define its *marker renaming graph*

$$\lfloor f \rfloor \stackrel{\rm def}{=} \mathit{oplus}(\mathit{return}_Y^{T^{\mathcal{G}}} \circ f) \in \mathcal{G}_{{\rm f}Y}^X.$$

Then, $\lfloor id_X \rfloor$ becomes the identity for the composition @.

Next, we define a monad morphism $\gamma\colon T \to T^{\mathcal{G}}$; for a finite set X and $s \in T(X)$, $\gamma(s) \stackrel{\rm def}{=} (\{*\}, B, \{\& \mapsto *\}) \in \mathcal{G}_{{\rm f}X}$ where $B(*) \stackrel{\rm def}{=} T(in_r)(s) \in T(\mathcal{L}_\epsilon \times \{\&\} + X)$. For example, for $T = P_{\rm fin}$ and $2 = \{\&0, \&1\} \in P_{\rm fin}(2)$, $\gamma(2) \in T^{\mathcal{G}}(2) = \mathcal{G}_{{\rm f}2}$ is a single node graph where the node has two output markers $\&0$ and $\&1$ and no edges.

Now, as promised above, let us consider $T^{\mathcal{G}}$ as an instance of T for the function (1). For $n = \{\&0, ..., \&n\text{–}1\}$ and $G \in T^{\mathcal{G}}(n) = \mathcal{G}_{{\rm f}n}$, its algebraic operation is

$$o(G)_Y\colon (\mathcal{G}_{{\rm f}Y})^n \to \mathcal{G}_{{\rm f}Y}$$
$$f \mapsto G \mathbin{@} \mathit{oplus}(f).$$

For example, for $T = P_{\rm fin}$ and $2 = \{\&0, \&1\} \in P_{\rm fin}(2)$, the binary operator

$$o(\gamma(2))_Y\colon (\mathcal{G}_{{\rm f}Y})^2 \to \mathcal{G}_{{\rm f}Y}$$
$$(G_0, G_1) \mapsto \gamma(2) \mathbin{@} (\&0 := G_0 \oplus \&1 := G_1) \qquad (2)$$

agrees with $\cup\colon (\mathcal{G}_{{\rm f}Y})^2 \to \mathcal{G}_{{\rm f}Y}$ in UnCAL. As explained above, $\cup$ for $\mathcal{G}_{{\rm f}Y}^X$ in Figure 4 is just the X-th power algebra of $\cup$ for $\mathcal{G}_{{\rm f}Y}$. Observe that, corresponding to @ and $\oplus$ occurring in (2), in Figure 4, the graph constructor $\cup$ has similar parts to @ and $\oplus$.

Let us sum up the above.

Definition 9 (T-Algebraic Graph Constructor). Let T be a finitary monad. For a generic effect $s \in T(n)$ ($n = \{\&0, ..., \&n\text{–}1\}$), the *$T$-algebraic graph constructor* $\mathbf{op}_s$ *of* s is defined as below.

For graphs $G_0, ..., G_{n-1} \in \mathcal{G}_{{\rm f}Y}^X$,

$$\mathbf{op}_s(G_0, ..., G_{n-1}) \stackrel{\rm def}{=} (\gamma(s) \mathbin{\_} X) \mathbin{@} \left(\bigoplus_{i=0}^{n-1} (\lfloor iso_i \rfloor \mathbin{@} G_i) \right) \in \mathcal{G}_{{\rm f}Y}^X$$

where

$$\gamma(s) \mathbin{\_} X \stackrel{\rm def}{=} \bigoplus_{\&x \in X} (\&x := (\gamma(s) \mathbin{@} \lfloor in_{\&x} \rfloor)) \in \mathcal{G}_{{\rm f}n \times X}^X$$
$$in_{\&x} \stackrel{\rm def}{=} \{\&i \mapsto (\&i, \&x)\}\colon n \to n \times X$$
$$iso_i \stackrel{\rm def}{=} \pi_r\colon \{\&i\} \times X \to X. \qquad \square$$

Signatures of Monads

In general, there are infinitely many generic effects $s \in T(n) (n = 1, 2, ...)$. For defining syntax of $\lambda_{\rm FG}^T$, we have to choose a set Σ of generic effects such that Σ is "representable in computer" and generates all generic effects.

For a finitary monad, there is a subset Σ of $\coprod_{n \in \mathbb{N}} T(n)$ such that, for every set S, any generic effect in $T(S)$ is a finitely many times iterated composition of some generic effects in Σ; here, for generic effects $s \in T(n) (n = \{0, ..., n\text{–}1\})$ and $t_i \in T(S) (i = 0, ..., n\text{–}1)$, their *composition* means the Kleisli composition $\mathit{lift}^T(i \mapsto t_i)(s) \in T(S)$. We call such Σ a *signature* and call an element s in $\Sigma \cap T(n)$ a *function symbol of arity* n. The whole set $\coprod_{n \in \mathbb{N}} T(n)$ itself is one (maximum) signature, but usually there is a much smaller signature—finite, or presentable in a meta language for implementation—as the following examples.

Example 10. When $T = P_{\rm fin}$, we can take a signature $\{\{\} (\in P_{\rm fin}(0)), \{0,1\} (\in P_{\rm fin}(2))\}$. Then, $\mathbf{op}_{\{\}}$ and $\mathbf{op}_{\{0,1\}}$ are the same as the graph constructors $\{\}$ and $\cup$ in UnCAL, respectively. The cases of $List$ and $M_{\rm fin}$—which correspond to monoid and commutative monoid, respectively—are quite similar.

For the finite probability monad $D_{\rm fin}$, let us see that the following Σ becomes a signature:

$$\Sigma(2) \stackrel{\rm def}{=} \{s_r\colon 2 \to [0,1] \mid r \in [0,1]\} \quad \subseteq D_{\rm fin}(2)$$
$$\Sigma(n) \stackrel{\rm def}{=} \emptyset \quad \text{(for other } n\text{)}$$

where s_r is defined as $s_r(0) \stackrel{\rm def}{=} r$ and $s_r(1) \stackrel{\rm def}{=} 1\text{–}r$. For $\phi_1, \phi_2\colon S \to [0,1]$ in $D_{\rm fin}(S)$, $o(s_r)(\phi_0, \phi_1)\colon S \to [0,1]$ in $D_{\rm fin}(S)$ is as below:

$$o(s_r)(\phi_0, \phi_1)(s) = r \cdot \phi_0(s) + (1\text{–}r) \cdot \phi_1(s).$$

Now recall that "singletons" in $D_{\rm fin}(S)$, i.e., elements in $\mathit{return}(S)$ are given as the Dirac delta functions δ_s as in Example 2. Then it is easy to see that the above Σ in fact becomes a signature: for example, $\phi\colon \mathbb{N} \to [0,1]$ in $D_{\rm fin}(\mathbb{N})$ such that

$$\phi(0) = \frac{1}{2}, \quad \phi(1) = \frac{1}{6}, \quad \phi(2) = \frac{1}{3}, \quad \phi(n) = 0 \text{ (for other } n\text{)}$$

can be represented by algebraic operations from the above Σ as

$$\phi = \frac{1}{2}\delta_0 + \frac{1}{6}\delta_1 + \frac{1}{3}\delta_2 = \frac{1}{2}\delta_0 + \frac{1}{2}(\frac{1}{3}\delta_1 + \frac{2}{3}\delta_2)$$
$$= o(s_{\frac{1}{2}})(\delta_0, o(s_{\frac{1}{3}})(\delta_1, \delta_2))$$
$$\text{(or, } = o(s_{\frac{1}{6}})(\delta_1, o(s_{\frac{3}{5}})(\delta_0, \delta_2)) \text{ etc.)}.$$

As this example, though a signature may be infinite, still may be presentable in a meta-language for implementation. $\square$

A signature is enough to represent all T-algebraic graph constructors: For generic effects $s \in T(n)(n{=}\{0,...,n{-}1\})$ and $t_i \in T(m)(i = 0,...,n{-}1)$, the T-algebraic graph constructor of the Kleisli composition $\mathit{lift}^T(i \mapsto t_i)(s) \in T(m)$ agrees with the composition of the corresponding T-algebraic graph constructors $\mathbf{op}_s : (\mathcal{G}_{\mathrm{f}Y}^X)^n \to \mathcal{G}_{\mathrm{f}Y}^X$ and $\mathbf{op}_{t_i} : (\mathcal{G}_{\mathrm{f}Y}^X)^m \to \mathcal{G}_{\mathrm{f}Y}^X$, i.e., $\mathbf{op}_s \circ \langle \mathbf{op}_{t_i} \rangle_{i\in n} : (\mathcal{G}_{\mathrm{f}Y}^X)^m \to \mathcal{G}_{\mathrm{f}Y}^X$. Hence, by the definition of signatures, any T-algebraic graph constructor can be represented as a finitely many times iterated composition of T-algebraic graph constructors of some function symbols in a signature.

Thus, though syntax of λ_{FG}^T depends on the choice of signatures of T, the expressive power of T-algebraic graph constructors is the same regardless of the choice.

As an important remark, the above definition of T-algebraic graph constructors gives us equational theories for T-algebraic graph constructors for free. T-algebraic graph constructors are defined through the monad morphism $\gamma\colon T \to T^{\mathcal{G}}$, so the T-algebraic graph constructors obey the same axioms as those of the algebra of T. For example, finite powersets are free algebras of upper semilattices; hence, the graph constructors $\{\}$ and $\cup$ satisfy all axioms of upper semilattices: i.e., associativity, unitality, commutativity, and idempotency. Moreover, since γ is monic, the converse, a kind of completeness, also holds.

3.2.3 Full Representability of Finite Graphs

The next proposition is the most important property of graph constructors.

Proposition 11 (Full Representability of Finite Graphs). *Let T be a finitary monad on* **Set** *which has an extension for ε-elimination. Any finite T-graph can be represented up to bisimilarity as finitely many applications of the graph constructors.* □

Here we give a simple proof of the above proposition using the notion of markers; this proof is a generalization with monads of an idea explained by example in [6]. However, the use of markers is not essential; for a naive and less simple proof, see [2, Appendix D].

Before the proof, we define *1-step unfolding* function

$$\mathit{uf}\colon \mathcal{G}_{\mathrm{f}Y} \to T(\mathcal{L}_\epsilon{\times}\mathcal{G}_{\mathrm{f}Y}{+}Y).$$

First, for a graph $G = (V, B, I) \in \mathcal{G}_{\mathrm{f}Y}^X$, and a node $v \in V$, we define

$$G|_v \stackrel{\text{def}}{=} (V, B, \{\&\mapsto v\}) \in \mathcal{G}_{\mathrm{f}Y} \tag{3}$$

thus, we have a function $G|_{(\text{-})}\colon V \to \mathcal{G}_{\mathrm{f}Y}$. Then for $G = (V, B, I) \in \mathcal{G}_{\mathrm{f}Y}$,

$$\mathit{uf}(G) \stackrel{\text{def}}{=} T(\mathcal{L}_\epsilon{\times}(G|_{(\text{-})}){+}Y)(B(I(\&))).$$

The function uf is strong bisimulation generic; taking the quotient $\mathcal{G}_{\mathrm{f}Y}/_{\sim_{\mathrm{s}}}$, the function

$$\mathit{uf}/_{\sim_{\mathrm{s}}}\colon \mathcal{G}_{\mathrm{f}Y}/_{\sim_{\mathrm{s}}} \to T(\mathcal{L}_\epsilon{\times}(\mathcal{G}_{\mathrm{f}Y}/_{\sim_{\mathrm{s}}}){+}Y)$$

induced from uf is nothing but the coalgebra structure of *final locally finite coalgebra* [1, 20]. Using [1, Theorem 3.3] and [20, Corollary III.15], we can show that for any finitary monad T, $\mathit{uf}/_{\sim_{\mathrm{s}}}$ is isomorphic.

Let us define the inverse-up-to-strong-bisimilarity of uf:

$$\mathit{uf}^{\text{-}1}\colon T(\mathcal{L}_\epsilon{\times}\mathcal{G}_{\mathrm{f}Y}{+}Y) \to \mathcal{G}_{\mathrm{f}Y},$$

i.e., $\mathit{uf}^{\text{-}1}$ is also strong bisimulation generic and the induced function $\mathit{uf}^{\text{-}1}/_{\sim_{\mathrm{s}}}$ becomes the inverse of $\mathit{uf}/_{\sim_{\mathrm{s}}}$. This can be constructed just by our graph constructors:

$$\mathit{uf}^{\text{-}1} = t \circ T([s, s'])$$

where

$$s \stackrel{\text{def}}{=} \langle(\text{-}):(\text{-})\rangle\colon \mathcal{L}_\epsilon{\times}\mathcal{G}_{\mathrm{f}Y} \to \mathcal{G}_{\mathrm{f}Y},$$
$$s' \stackrel{\text{def}}{=} \langle\text{-}\rangle\colon Y \to \mathcal{G}_{\mathrm{f}Y},$$

and

$$t \stackrel{\text{def}}{=} \mu_Y^{T^{\mathcal{G}}} \circ \gamma_{\mathcal{G}_{\mathrm{f}Y}} : T(\mathcal{G}_{\mathrm{f}Y}) \to \mathcal{G}_{\mathrm{f}Y} \tag{4}$$
$$(\mu_Y^{T^{\mathcal{G}}} \stackrel{\text{def}}{=} \mathit{lift}^{T^{\mathcal{G}}}(\mathit{id}_{T^{\mathcal{G}}(Y)})).$$

Note that since T is finitary, for any element $x \in T(\mathcal{G}_{\mathrm{f}Y})$, there exist $s \in T(n)$ and n-number of graphs G_i—i.e., $G_{(\text{-})}\colon n \to \mathcal{G}_{\mathrm{f}Y}$—such that $x = T(G_{(\text{-})})(s)$. Then,

$$\begin{aligned}
t(x) &= (\mu_Y^{T^{\mathcal{G}}} \circ \gamma_{\mathcal{G}_{\mathrm{f}Y}} \circ T(G_{(\text{-})}))(s) && \text{(naturality of } \gamma)\\
&= (\mu_Y^{T^{\mathcal{G}}} \circ T^{\mathcal{G}}(G_{(\text{-})}) \circ \gamma_n)(s) && \text{(by monad axioms)}\\
&= (\mathit{lift}^{T^{\mathcal{G}}}(G_{(\text{-})}) \circ \gamma_n)(s) && \text{(def. of } \mathit{lift}^{T^{\mathcal{G}}})\\
&= \gamma_n(s) \mathbin{@} \mathit{oplus}(G_{(\text{-})}) && \text{(def. of } \mathbf{op})\\
&= \mathbf{op}_s(G_0, ..., G_{n-1}).
\end{aligned}$$

Hence $t(x)$ is defined with the T-algebraic graph constructors. For example, when $T = \mathit{List}$,

$$t = \mathbf{foldr}(+\!\!+, [])\colon \mathit{List}(\mathcal{G}_{\mathrm{f}Y}) \to \mathcal{G}_{\mathrm{f}Y}.$$

On the above s, precisely, $s(\varepsilon, G) = \langle \varepsilon : G\rangle$ is not a representation by graph constructors, because an expression $\langle a : e\rangle$ is not allowed in λ_{FG}^T when $a = \varepsilon$. For the case, in the above definition, replace such $\langle \varepsilon : e\rangle$ with e, which is bisimilar to $\langle \varepsilon : e\rangle$.

Now let us give the proof of Proposition 11.

PROOF. Let $G = (V, B, I)$ be a finite graph in $\mathcal{G}_{\mathrm{f}Y}^X$. First we prepare a marker $\&v$ for each node v, then we write $\&V$ for the set of the markers, and define $f\colon V \to \&V$ as $f(v) = \&v$. Then for each v,

$$G_v \stackrel{\text{def}}{=} \mathit{uf}^{\text{-}1}\Big(T\big(\mathcal{L}_\epsilon{\times}(\langle\text{-}\rangle \circ f){+}Y\big)\big(B(v)\big)\Big) \in \mathcal{G}_{\&V+Y}$$

can be represented by graph constructors. Then

$$\lfloor f \circ I \rfloor \mathbin{@} \mathbf{cycle}\Big(\mathop{\oplus}_{v\in V} \&v := G_v\Big) \in \mathcal{G}_Y^X$$

is bisimilar to the original graph G. □

For example, for the graph G in Example 1,

$$\begin{aligned}
G_1 &= \langle d : \langle \&2\rangle\rangle +\!\!+ \langle a : \langle \&4\rangle\rangle\\
G_2 &= \langle c : \langle \&3\rangle\rangle\\
G_3 &= \langle d : \langle \&2\rangle\rangle\\
G_4 &= \langle b : \langle \&3\rangle\rangle +\!\!+ \langle \&y\rangle,
\end{aligned}$$

then G is bisimilar to

$$\langle \&1\rangle \mathbin{@} \mathbf{cycle}\big((\&1{:=}G_1) \oplus (\&2{:=}G_2) \oplus (\&3{:=}G_3) \oplus (\&4{:=}G_4)\big).$$

3.2.4 The Need of Monad Extensions for ε-elimination

So far, we have seen the importance of the *finitarity* of a monad T for *finite*-graph transformation calculus λ_{FG}^T. Here, we explain why we need to care with the phenomenon of occurring infinite width in this paper and why we did not need to take such special care in the case of the original UnCAL for P_{fin}-graphs.

When T is finitary and has an extension T' for ε-elimination, if we want to define ε-elimination only for *finite* T-graphs, without loss of generality, we can replace the extension T' with its *finitary part* $T'|_{\mathrm{fin}}$:

$$T'|_{\mathrm{fin}}(S) \stackrel{\text{def}}{=} \bigcup_{S'\subseteq S,\ S'\colon \text{finite}} T'(S').$$

(See Appendix A for details.)

For unordered graphs, $(P_{\mathrm{cnt}})|_{\mathrm{fin}}$ is equal to P_{fin}; thus, we do not need P_{cnt} and ε-elimination does not produce infinite width. This is the reason why we did not need to consider infinite width for UnCAL.

For ordered graphs, $(CList)|_{\mathrm{fin}}(S)$ consists of such countable lists l that the number of elements of S that occur in l are finite. Hence, still we can not avoid the notion of a countable linear order, i.e., infinite width, for even finite ordered graphs having ε-edges. (For the details on ordered graphs, see [14, 2].)

Also on the other two examples M_{fin} and D_{fin}, $T'|_{\mathrm{fin}}$ does not become the original T: i.e.,

$$(M_{\mathrm{cnt}})|_{\mathrm{fin}}(S) = \{\phi\colon S \to \mathbb{N}\cup\{\infty\} \mid \phi^{-1}(\mathbb{N}-\{0\}) \text{ is finite}\}$$
$$(SubD_{\mathrm{cnt}})|_{\mathrm{fin}}(S) = \{\phi\colon S \to [0,1] \mid \phi^{-1}((0,1]) \text{ is finite}, \Sigma_x \phi(x) \leq 1\}.$$

However, in these cases, $T'|_{\mathrm{fin}}$ are easily representable on computers differently from the case of infinite-width ordered graphs.

3.3 Structural Recursion

Now, let us define structural recursion for T-graphs, which plays a leading role in transformations of λ^T_{FG}. We again show the typing rule for structural recursion:

$$\frac{\Gamma,\, l{:}\mathbf{Label},\, g{:}\mathbf{G}_Y \vdash e\colon \mathbf{G}^Z_Z \qquad \Gamma \vdash e'\colon \mathbf{G}^X_Y}{\Gamma \vdash \mathbf{srec}(\lambda(l,g).e)(e')\colon \mathbf{G}^{Z\times X}_{Z\times Y}}$$

When $T = P_{\mathrm{fin}}$, the structural recursion $f = \mathbf{srec}(\lambda(l,g).e)$ satisfies the following characteristic equations, where we consider single-rooted case for simplicity.

$$\begin{array}{lcl} f(\{\}) & = & \{\} \\ f(g_1 \cup g_2) & = & f(g_1) \cup f(g_2) \\ f(\langle l : g\rangle) & = & e(l,g)\ @\ f(g) \\ f(\langle \&y\rangle) & = & \lfloor \&z \mapsto (\&z, \&y)\rfloor. \end{array}$$

By regarding the above equations as an recursive definition for infinite trees, the above equations serve as a definition of f to be a function which inputs *finite* graphs and outputs *infinite* graphs. However, the outputs of f are in fact (bisimilar to) *finite* graphs, and this is proved by using the following bulk semantics of f.

Now, we give a definition of the bulk semantics, explaining with Figure 5. In Figure 5, in each step, gray parts show parts unchanged from the previous step. A marker above a node is an input marker and markers below a node mean output markers. Each box with bold frame is itself a graph, while each dotted box—a part of a graph—has no meaning, just hints that it was a graph before.

Briefly explaining, with bulk semantics, we first calculate the application of a given input function e to each pair of an edge and its following subgraph, as the graph (b) in Figure 5; though the example of e in the figure does not use the second argument for simplicity of explanation. After that, we connect the results in keeping with the shape of the original graph by using ε-edges, which results in the graph (f) in Figure 5.

Definition 12 (Bulk Semantics of Structural Recursion). Let T be a finitary monad on $\mathbf{Set}$. For a function $e\colon \mathcal{L}\times\mathcal{G}_{\mathrm{f}Y} \to \mathcal{G}_{\mathrm{f}Z}$, a *structural recursion function* $\mathbf{srec}(e)\colon \mathcal{G}_{\mathrm{f}Y} \to \mathcal{G}_{\mathrm{f}Z}$ is defined through the following steps. Before that, we extend e to $\bar{e}\colon \mathcal{L}_\epsilon\times\mathcal{G}_{\mathrm{f}Y} \to \mathcal{G}_{\mathrm{f}Z}$ which maps additionally $(\varepsilon, \_)$ to the "identity graph" $\lfloor id_Z \rfloor \in \mathcal{G}_{\mathrm{f}Z}$ (see the upper leftmost picture in Figure 5). In the following, the steps from (a) to (f) correspond to those in Figure 5.

(a) **[Input of $\mathbf{srec}(e)$].** Let us take $G = (V, B, I) \in \mathcal{G}_{\mathrm{f}Y}$.

(b) **[Relabeling by e].** Let us define $\bar{e}\colon \mathcal{L}_\epsilon\times V \to \mathcal{G}_{\mathrm{f}Z}$ as $\bar{e} \circ (id_{\mathcal{L}_\epsilon}\times G|_{(\text{-})})$, where $G|_{(\text{-})}\colon V \to \mathcal{G}_{\mathrm{f}Y}$ is defined as Equation (3). Then, we construct a new branching function B_1 whose labels are graphs in $\mathcal{G}_{\mathrm{f}Z}$: B_1 is defined as the composition of

$$V \xrightarrow{B} T(\mathcal{L}_\epsilon\times V+Y) \xrightarrow{T(\langle\bar{e},\pi_2\rangle+id_Y)} T(\mathcal{G}_{\mathrm{f}Z}\times V+Y).$$

(c) **[Storing target nodes in output markers and duplicating original output markers].** Let B_2 be the composition of the following functions

$$V \xrightarrow{B_1} T(\mathcal{G}_{\mathrm{f}Z}\times V+Y) \xrightarrow{T([p,q])} T(\mathcal{G}_{\mathrm{f}Z\times V+Z\times Y})$$

where p and q are the obvious functions:

$$p(g,v) \stackrel{\text{def}}{=} g\ @\ \lfloor \&z \mapsto in_l((\&z, v))\rfloor$$
$$q(\&y) \stackrel{\text{def}}{=} \lfloor \&z \mapsto in_r((\&z, \&y))\rfloor.$$

(d) **[Connecting horizontally by T-algebraic graph constructors].** Now $\mathcal{G}_{\mathrm{f}Y}$ has the algebraic structure t_Y defined in Equation (4) after Proposition 11, and we can regard input marker sets as power through $oplus\colon (\mathcal{G}_{\mathrm{f}Y})^X \xrightarrow{\cong} \mathcal{G}_{\mathrm{f}Y}$, so we have the obvious power algebra structure $t^{(X)}_Y$ on $\mathcal{G}_{\mathrm{f}Y}$, i.e., the composition of

$$T(\mathcal{G}_{\mathrm{f}Y}) \to T(\mathcal{G}_{\mathrm{f}Y})^X \xrightarrow{(t_Y)^X} \mathcal{G}_{\mathrm{f}Y}.$$

Then, let B_3 be the composition of

$$V \xrightarrow{B_2} T(\mathcal{G}_{\mathrm{f}Z\times V+Z\times Y}) \xrightarrow{t^{(Z)}_{Z\times V+Z\times Y}} \mathcal{G}_{\mathrm{f}Z\times V+Z\times Y}.$$

(e) **[Disjoint union by $\oplus$].** Using the bijective correspondence

$$oplus : (V\to\mathcal{G}_{\mathrm{f}Z\times V+Z\times Y}) \xrightarrow{\cong} \mathcal{G}_{\mathrm{f}Z\times V+Z\times Y},$$

we obtain a graph $G' \stackrel{\text{def}}{=} oplus(B_3) \in \mathcal{G}_{\mathrm{f}Z\times V+Z\times Y}$.

(f) **[Connecting vertically by cycle and input marker renaming].** Finally, we define

$$\mathbf{srec}(e)(G) \stackrel{\text{def}}{=} \lfloor id_Z\times I\rfloor @ \mathbf{cycle}(G') \quad (\in \mathcal{G}_{\mathrm{f}Z\times Y}).$$

□

We remark that the finitarity of a monad T is used at the step (d) for the algebra t_Y. The above definition works if we replace all $\mathcal{G}_{\mathrm{f}}$ with $\mathcal{G}$; then, T does not need to be finitary.

The above semantics can be implemented in an obvious way (a hint for implementation can be found in [14]). Our implementation for unordered graphs and ordered graphs in OCaml can be found at `http://www.biglab.org/src/lambdaFG/`.

For generic implementation parameterizing monads, in a meta language we define a class of monads-with-signatures with three methods: a signature $g\colon \Sigma\to T(\mathbb{N})$, an arity function $a\colon \Sigma\to\mathbb{N}$, and $\mathrm{fold}\colon \forall S.(\prod_{s\in\Sigma}(S^{a(s)}\to S))\to(T(S)\to S)$. In, e.g., Haskell, the dependent type $\prod_{s\in\Sigma}(S^{a(s)}\to S)$ can be replaced with a larger type $\Sigma\to(S^{\mathbb{N}}\to S)$, since there is a right inverse of the projection $S^{\mathbb{N}}\to S^{a(s)}$, i.e., $id_{S^{a(s)}}\times\bot\colon S^{a(s)}\times 1 \to S^{a(s)}\times S^{\mathbb{N}} \cong S^{\mathbb{N}}$.

Finally, we give an example of queries using the structural recursion; for more examples, see [6, 15].

Example 13. Consider an ordered graph representation of books. Since "sections" are ordered and there are some reference links in books, we can see books as ordered graphs. The following structural recursion toc, which is adapted from [25], computes the table

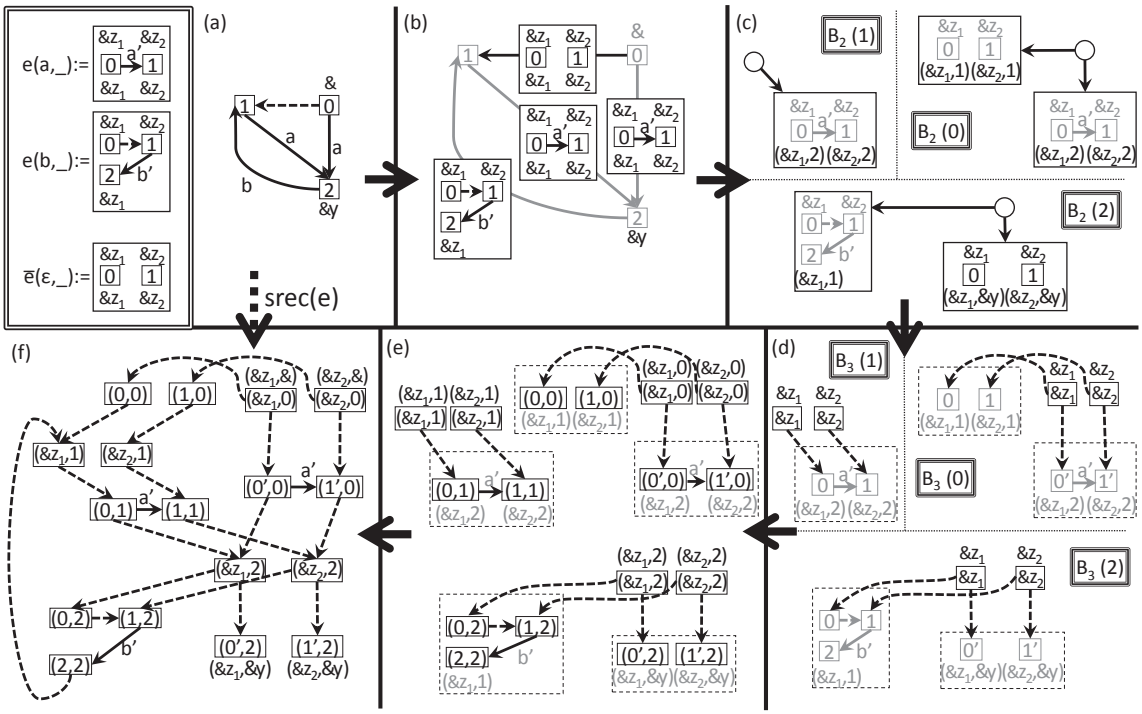


Figure 5: Bulk Semantics of Structural Recursion: Example with Ordered Graphs

of contents of books in which sections can be arbitrarily nested:

$$toc(db) = \mathbf{srec}(\ \lambda(l,g).\,\mathbf{if}\ l = \texttt{section}\ \mathbf{then}\ \langle \texttt{section} : (get\_title(g) +\!\!+ \langle \& \rangle)\rangle\ \mathbf{else}\ \langle \& \rangle\ )\ (db)$$

where the function get_title results in the title of the section:

$$get\_title(g) = \mathbf{srec}(\ \lambda(l_1,g_1).\,\mathbf{if}\ l_1 = \texttt{title}\ \mathbf{then}\ \langle \texttt{title} : \mathbf{srec}(\ \lambda(l_2,g_2).\langle l_2 : [] \rangle\ )\ (g_1)\rangle\ \mathbf{else}\ []\ )\ (g)$$

□

3.4 Bisimilarity for Higher Order Functions

So far we have given the semantics with (V, B, I) form at the equality level. In the rest of this section, we consider semantics at the bisimilarity level.

In Section 2, we gave the semantic equivalence for graph types $\mathbf{G}_Y^X$, i.e., the bisimilarity. Since λ_{FG}^T has higher order functions, and **srec** is a higher order function, we have to extend the semantic equivalence for base types to function types. It is well known that if we lift an equivalence relation to function types we need to switch from the notion of an equivalence relation to that of a *partial equivalence relation*, i.e., an equivalence relation on some subset of the original set. This is because, now, not all functions on $\mathcal{G}_{\mathrm{f}Y}^X$ are bisimulation generic, so we have to cut out the *subset* consisting of bisimulation generic functions.

Let us give the formal definition. For the types σ of λ_{FG}^T, we define binary logical relations $\sim_\sigma$ on $[\![\sigma]\!]$. When $\sigma = \mathbf{G}_Y^X$, $\sim_{\mathbf{G}_Y^X}$ is the bisimilarity on $\mathcal{G}_{\mathrm{f}Y}^X$; for the other base types σ, $\sim_\sigma$ are just the equality relations. When $\sigma = \sigma_1 \times \sigma_2$, we define a binary relation $\sim_\sigma$ on $[\![\sigma]\!] \stackrel{\text{def}}{=} [\![\sigma_1]\!] \times [\![\sigma_2]\!]$ as

$$(x_1, x_2) \sim_\sigma (x_1', x_2') \stackrel{\text{def}}{\Longleftrightarrow} (x_1 \sim_{\sigma_1} x_1') \wedge (x_2 \sim_{\sigma_2} x_2').$$

When $\sigma = \sigma_1 \to \sigma_2$, we define a binary relation $\sim_\sigma$ on $[\![\sigma]\!] \stackrel{\text{def}}{=} [\![\sigma_1]\!] \to [\![\sigma_2]\!]$ as

$$f \sim_\sigma f' \stackrel{\text{def}}{\Longleftrightarrow} \forall x, x' \in [\![\sigma_1]\!].\ (x \sim_{\sigma_1} x' \Rightarrow f(x) \sim_{\sigma_2} f'(x')).$$

Then for any type σ, $\sim_\sigma$ becomes a partial equivalence relation on $[\![\sigma]\!]$, i.e., an equivalence relation on the subset

$$|\sim_\sigma| \stackrel{\text{def}}{=} \{x \in [\![\sigma]\!] \mid x \sim_\sigma x\}.$$

We call a function $f\colon [\![\sigma_1]\!] \to [\![\sigma_2]\!]$ *(higher order) bisimulation generic* if f is in $|\sim_{\sigma_1 \to \sigma_2}|$. (Note that this kind of lifting of semantic equivalence to function types is possible for any equivalence relation such as strong bisimilarity, graph isomorphism etc.)

Then from the Basic Lemma of logical relation, interpretations of all the terms are bisimulation generic if interpretations of all the constants are bisimulation generic; as a result of the consequent of this implication, we obtain a model of λ_{FG}^T in the cartesian closed category **Set**. (See the textbook [22] for the technique of logical relation.) In the rest of this paper, we prove that the interpretations of all the constants of λ_{FG}^T are bisimulation generic.

3.5 Bisimulation Genericity of Terms of λ_{FG}^T

First we show the bisimulation genericity of the graph constructors.

Proposition 14 (Bisimulation Genericity of Graph Constructors). *Let T be a finitary monad having an extension for ε-elimination. All the graph constructors are bisimulation generic.* □

PROOF. For a constructor f, prove that $\varepsilon\text{-elim}(f(G_1, ..., G_n))$ is strongly bisimilar to $\varepsilon\text{-elim}(f(\varepsilon\text{-elim}(G_1), ..., \varepsilon\text{-elim}(G_n)))$. This reduces the bisimulation genericity in the statement to strong-bisimulation genericity, which is obvious. □

Next, let us consider structural recursion. The result below is stronger than what is proved in [6], even when $T = P_{\mathrm{fin}}$, because

here bisimulation genericity is proved also on the first argument e, while in [6], it is proved only on the second argument G. This is the key point of our extension from UnCAL to the higher order calculus $\lambda_{\mathrm{FG}}^{P_{\mathrm{fin}}}$.

Theorem 15 (Bisimulation Genericity of Structural Recursion). *Let T be a finitary monad having an extension for ε-elimination T'. Structural recursion* $\mathbf{srec}$ *is bisimulation generic, i.e., if*

$$e_1 \sim e_2 \colon \mathcal{L}\times T\text{-}\mathcal{G}_{\mathrm{f}Y} \to T\text{-}\mathcal{G}_{\mathrm{f}Z}^{\,Z}, \quad \textit{and} \quad G_1 \sim G_2 \in T\text{-}\mathcal{G}_{\mathrm{f}Y}^{\,X},$$

then

$$\mathbf{srec}(e_1)(G_1) \sim \mathbf{srec}(e_2)(G_2) \in T\text{-}\mathcal{G}_{\mathrm{f}Z\times Y}^{\,Z\times X}.$$

PROOF. Basically we want to prove in a similar way to the proof of Proposition 14, but there is a bit subtle problem. Though the following might be expected to hold,

$$\varepsilon\text{-elim}(\mathbf{srec}(e)(G)) \ \sim_{\mathrm{s}} \ \varepsilon\text{-elim}(\mathbf{srec}(\varepsilon\text{-elim}\circ e)(\varepsilon\text{-elim}(G)))$$

in fact, there is a type error: i.e., now $\varepsilon\text{-elim}\circ e$ is a function of the type $\mathcal{L}\times T\text{-}\mathcal{G}_{\mathrm{f}Y} \to T'\text{-}\mathcal{G}_{\mathrm{f}Z}^{\,Z}$ so we can not apply $\mathbf{srec}$ with respect to either T nor T'. This can be solved as follows.

First, it is easily shown that the structural recursion is defined "uniformly" on monads: Let

$$i \stackrel{\text{def}}{=} \iota\text{-}\mathcal{G}_{\mathrm{f}Y}^{\,X} \colon T\text{-}\mathcal{G}_{\mathrm{f}Y}^{\,X} \to T'\text{-}\mathcal{G}_{\mathrm{f}Y}^{\,X},$$

where $\iota\text{-}\mathcal{G}_{\mathrm{f}Y}^{\,X}$ is defined as just the restriction of $\iota\text{-}\mathcal{G}_{Y}^{X}$. For

$$e \colon \mathcal{L}\times T\text{-}\mathcal{G}_{\mathrm{f}Y} \to T\text{-}\mathcal{G}_{\mathrm{f}Z}^{\,Z},$$

$$e' \colon \mathcal{L}\times T'\text{-}\mathcal{G}_{\mathrm{f}Y} \to T'\text{-}\mathcal{G}_{\mathrm{f}Z}^{\,Z},$$

$$G \in T\text{-}\mathcal{G}_{\mathrm{f}Y}^{\,X},$$

if $e'\circ(\mathcal{L}\times i) = i\circ e$ then

$$\mathbf{srec}^{T'}(e')(i(G)) = i(\mathbf{srec}^{T}(e)(G)).$$

Here the equality $=$ means the exact equality rather than strong bisimilarity or bisimilarity.

This reduces the setting of the theorem to the case in which T is not necessarily finitary and $\iota\colon T \to T'$ is the identity—i.e., T $(= T')$ is an arbitrary monad whose Kleisli category has an iteration operator—, so that ε-elim is closed in $T\text{-}\mathcal{G}_{\mathrm{f}Y}^{\,X}$. This is because, for any

$$e_1 \sim e_2 \colon \mathcal{L}\times T\text{-}\mathcal{G}_{\mathrm{f}Y} \to T\text{-}\mathcal{G}_{\mathrm{f}Z}^{\,Z},$$

there are

$$e_1' \sim e_2' \colon \mathcal{L}\times T'\text{-}\mathcal{G}_{\mathrm{f}Y} \to T'\text{-}\mathcal{G}_{\mathrm{f}Z}^{\,Z}$$

such that $e_i'\circ(\mathcal{L}\times i) = i\circ e_i$. We can take such e_i' by using some bisimulation generic retraction i^* of i. Such i^* can be defined in an ad hoc way[3]: Let G_0 be an arbitrary fixed graph in $T\text{-}\mathcal{G}_{\mathrm{f}Y}$. For $G' \in T'\text{-}\mathcal{G}_{\mathrm{f}Y}$, if G' is in $T\text{-}\mathcal{G}_{\mathrm{f}Y}$, then $i^*(G') \stackrel{\text{def}}{=} G'$, else if G' is bisimilar to some graph G in $T\text{-}\mathcal{G}_{\mathrm{f}Y}$, then $i^*(G') \stackrel{\text{def}}{=} G$ (G should be chosen), else $i^*(G') \stackrel{\text{def}}{=} G_0$.

Now, in this reduced setting, we will prove the goal in a similar way to Proposition 14. First, we can prove the commutativity of $\mathbf{srec}$ with ε-elim:

$$\varepsilon\text{-elim}(\mathbf{srec}(e)(G)) \ \sim_{\mathrm{s}} \ \varepsilon\text{-elim}(\mathbf{srec}(\varepsilon\text{-elim}\circ e)(\varepsilon\text{-elim}(G))).$$

Then, for concluding the proof, it is enough to show the strong-bisimulation genericity of $\mathbf{srec}$ (Lemma 16). □

[3] For one who wants non-ad-hoc way, if we assume that the given function e_i is the interpretation of some term t_i in $\lambda_{\mathrm{FG}}^{T}$, then we can get the desired e_i' as the interpretation of t_i in $\lambda_{\mathrm{FG}}^{T'}$.

Lemma 16. *The structural recursion* $\mathbf{srec}$ *is strong-bisimulation generic: i.e., for $e_1 \sim_{\mathrm{s}} e_2\colon \mathcal{L}\times\mathcal{G}_{\mathrm{f}Y} \to \mathcal{G}_{\mathrm{f}Z}^{\,Z}$ and for $G_1 \sim_{\mathrm{s}} G_2 \in \mathcal{G}_{\mathrm{f}Y}^{\,X}$, the following holds:*

$$\mathbf{srec}(e_1)(G_1) \sim_{\mathrm{s}} \mathbf{srec}(e_2)(G_2) \ \ (\in \ \mathcal{G}_{\mathrm{f}Z\times Y}^{\,Z\times X}).$$

PROOF. Basically, this is proved straightforwardly according to the steps in Definition 12. At the step (f) in Definition 12, we used $\mathbf{cycle}$; accordingly in this proof, we use the uniformity of the iteration operator. For details, see Appendix B. □

4. CONCLUDING REMARK

We presented a parameterized calculus $\lambda_{\mathrm{FG}}^{T}$ that is an extension of the lambda calculus with finite graph types, the graph constructors, and the structural recursion, for generalizing and extending UnCAL [6]. We presented the semantics of $\lambda_{\mathrm{FG}}^{T}$ that has a suitable bisimilarity accommodating ε-edges as well as the termination and finiteness preserving properties. A further extension of $\lambda_{\mathrm{FG}}^{T}$ is presented in [2].

This paper is based on our previous work [14]. As explained in the introduction, in the previous paper a new calculus—named as λ_{FG} in the previous paper—for ordered graphs was introduced. The calculus λ_{FG} is not the same as $\lambda_{\mathrm{FG}}^{List}$ in the current paper; here, we give the comparison of the current and the previous papers. In λ_{FG}, the structural recursion is extended from that in the current $\lambda_{\mathrm{FG}}^{List}$, so that it can transform sibling direction of graphs; hence the expressive power of λ_{FG} is higher than $\lambda_{\mathrm{FG}}^{List}$. Therefore to extend current work with such sibling transformations is important future work. Also, in the previous work we focused on ordered graphs, and gave some decidability results on ordered graphs. Moreover, in the current paper the existence of a monad and an iteration operator is used as an assumption; while, in the previous paper to define the countable list monad and the iteration operator for the monad was a main contribution.

The database community has undertaken a lot of work on graph transformation languages, but most of it has been on graphs up to isomorphism (or equality); there has been little work on graphs up to bisimilarity with considering bisimulation genericity. In [3], however, new semantics for the structural recursion of UnCAL was given.

There has been a lot of work on structural recursion for specific kinds of graphs, such as graphs represented by trees with specific pointers [11, 8], and graphs represented by trees with embedded functions [9, 7]. However, they do not ensure all of the bisimulation genericity, terminating property, and finiteness preserving property, which are our original goals as explained in the introduction.

In coalgebra theory, i.e., the study of infinitary/cyclic structures, some studies have focused on the finiteness of graphs. In [5, 28], for every Kripke polynomial endofunctor or quantitative functor F, a systematic way of giving a syntax fully representing all finite F-coalgebras and a sound and complete equational theory for bisimilarity were given. The differences between that study and ours are as follows: (i) The two classes of endofunctors—quantitative functor and ours with arbitrary finitary monads—are not comparable; especially, our leading example $List(\mathcal{L}_\epsilon\times(\text{-})+Y)$ is not a quantitative functor. (ii) Our equational theory is restrictive while the equational theory given in that study captures completely the bisimilarity. (iii) Their study does not treat ε-edges, and is not a study for transformations. (iv) The approaches to syntax are different: The both can express arbitrary finite coalgebras, but there seems no *compositional* translations between their and our graph representation systems, and comparison of expressive power of open terms is not obvious. Also, their approach is in a guarded style, by which

the unique fixed point operator can be used; while ours does not require guardedness, and require an iteration operator instead.

In [1, 20], the authors studied categorical properties of the set of finite coalgebras, and characterized it as a *final locally finite coalgebra*. The class of endofunctors for coalgebras in that study is wider than ours; some of the results are applied and inspire our work. In that study, there is no consideration for finiteness-preserving structural recursion. The finality among locally finite coalgebras is a kind of corecursion and has a similar problem to that of corecursion; i.e., to assure finiteness-preservation, we have to check local-finiteness of infinite graphs, automation of which seems difficult.

The general treatment for ε-edges with iteration operators in the current paper was hinted at in [16]. Although there was no consideration of ε-edges itself in that study, the author showed that the trace semantics for some kinds of coalgebra induces iteration operators by forgetting the length of trace paths; the resulting iteration operators can be regarded as ε-elimination. The countable list monad is not treated in that paper; it does not satisfy the assumption of the theorem in that paper.

In [19], the authors studied ε-elimination for weighted automata. The kinds of automata treated in their paper are parametrized by certain semirings, and do not include our ordered case ($List$-graphs). Also, their approach to specifying weighted automata is different from ours. They define first some class of weighted ε-automata, and then they restrict it to "valid" ones with some procedure; while our graphs are "valid" from the beginning. As a main contribution, they give an algorithm for removing ε-transitions. It seems interesting future work to compare their and our general automaton/graph models and merge each advantages.

5. ACKNOWLEDGMENTS

We thank Ichiro Hasuo and Kazutaka Matsuda for useful comments. The research was supported in part by the Grand-Challenging Project on the "Linguistic Foundation for Bidirectional Model Transformation" of the National Institute of Informatics and KAKENHI No. 23700047, 22300012, 25240009, and 23220001.

6. REFERENCES

[1] J. Adámek, S. Milius, and J. Velebil. Iterative algebras at work. *Mathematical. Structures in Comp. Sci.*, 16(6):1085–1131, Dec. 2006.

[2] K. Asada, S. Hidaka, H. Kato, Z. Hu, and K. Nakano. Parameterized graph transformation languages with monads. Technical Report GRACE-TR-2012-07, GRACE Center, National Institute of Informatics, 2012.

[3] A. A. Benczúr and B. Kósa. Static analysis of structural recursion in semistructured databases and its consequences. In *ADBIS*, volume 3255 of *LNCS*, pages 189–203. Springer, 2004.

[4] N. Benton, J. Hughes, and E. Moggi. Monads and effects. In *In International Summer School on Applied Semantics, APPSEM 2000*, pages 42–122. Springer-Verlag, 2000.

[5] M. Bonsangue, J. Rutten, and A. Silva. Algebras for Kripke polynomial coalgebras. In *LICS*, IEEE, Computer Science Press, pages 49–58, 2009.

[6] P. Buneman, M. F. Fernandez, and D. Suciu. UnQL: a query language and algebra for semistructured data based on structural recursion. *The VLDB Journal*, 9(1):76–110, 2000.

[7] B. C. d. S. Oliveira and W. R. Cook. Functional programming with structured graphs. In *ICFP*, pages 77–88. ACM Press, 2012.

[8] S. Dal Zilio, D. Lugiez, and C. Meyssonnier. A logic you can count on. In *POPL'04*, pages 135–146. ACM Press, 2004.

[9] L. Fegaras and T. Sheard. Revisiting catamorphisms over datatypes with embedded functions (or, programs from outer space). In *POPL*, pages 284–294. ACM Press, 1996.

[10] E. Haghverdi. *A Categorical Approach to Linear Logic, Geometry of Proofs and Full Completeness*. PhD thesis, University of Ottawa, 2000.

[11] M. Hamana. Initial algebra semantics for cyclic sharing structures. In *TLCA*, LNCS 5608, pages 127–141, 2009.

[12] M. Hasegawa. The uniformity principle on traced monoidal categories. *Electr. Notes Theor. Comput. Sci.*, 69:137–155, 2002.

[13] I. Hasuo, 2011. personal communication.

[14] S. Hidaka, K. Asada, Z. Hu, H. Kato, and K. Nakano. Structural recursion for querying ordered graphs. In *ICFP 2013, to appear*, Mar. 2013.

[15] S. Hidaka, Z. Hu, K. Inaba, H. Kato, K. Matsuda, and K. Nakano. Bidirectionalizing graph transformations. In *ICFP 2010*, pages 205–216. ACM Press, 2010.

[16] B. Jacobs. From coalgebraic to monoidal traces. *Electr. Notes Theor. Comput. Sci.*, 264(2):125–140, 2010.

[17] Y. Kakutani. Duality between call-by-name recursion and call-by-value iteration. In *CSL*, volume 2471 of *LNCS*, pages 506–521. Springer, 2002.

[18] G. M. Kelly. Structures defined by finite limits in the enriched context, I. *Cahiers Topologie Géom. Différentielle*, 23(1):3–42, 1982.

[19] S. Lombardy and J. Sakarovitch. The removal of weighted ϵ-transitions. In *Implementation and Application of Automata*, volume 7381 of *LNCS*, pages 345–352. Springer, 2012.

[20] S. Milius. A sound and complete calculus for finite stream circuits. In *LICS*, pages 421–430. IEEE Computer Society, 2010.

[21] R. Milner. *Communicating and Mobile Systems: the π-calculus*. Cambridge University Press, 1999.

[22] J. C. Mitchell. *Foundations for programming languages*. Foundation of computing series. MIT Press, 1996.

[23] E. Moggi. Computational lambda-calculus and monads. In *LICS*, pages 14–23. IEEE Computer Society, 1989.

[24] G. Plotkin and J. Power. Algebraic operations and generic effects. *Applied Categorical Structures*, 11:69–94, 2003.

[25] E. L. Robertson, L. V. Saxton, D. V. Gucht, and S. Vansummeren. Structural recursion as a query language on lists and ordered trees. *Theory of Computing Systems*, 44(4):590–619, 2009.

[26] J. Rutten. Universal coalgebra: a theory of systems. *Theoretical Computer Science*, 249(1):3 – 80, 2000.

[27] I. Sasano, Z. Hu, S. Hidaka, K. Inaba, H. Kato, and K. Nakano. Toward bidirectionalization of ATL with GRoundTram. In *ICMT*, volume 6707 of *LNCS*, pages 138–151. Springer, 2011.

[28] A. Silva, F. Bonchi, M. M. Bonsangue, and J. J. M. M. Rutten. Quantitative Kleene coalgebras. *Inf. Comput.*, 209(5):822–849, 2011.

[29] A. K. Simpson and G. D. Plotkin. Complete axioms for categorical fixed-point operators. In *LICS*, pages 30–41. IEEE Computer Society, 2000.

[30] A. Sokolova. *Coalgebraic Analysis of Probabilistic Systems*. PhD thesis, TU Eindhoven, 2005.

APPENDIX

A. FINITARY MONAD EXTENSION FOR ε-ELIMINATION

For a monad T' with an iteration operator $iter$ in $\mathbf{Set}_{T'}$, $T'|_{\mathrm{fin}}$ may not have an iteration operator (as $P_{\mathrm{cnt}}|_{\mathrm{fin}} = P_{\mathrm{fin}}$). Still, we can define ε-elimination for finite $T'|_{\mathrm{fin}}$-graphs so that the result graph is equal to one obeying Definition 5.

Let $G = (V, B, I)$ be a finite $T'|_{\mathrm{fin}}$-graph in $T'|_{\mathrm{fin}}$-$\mathcal{G}_{\mathrm{f}Y}^{\ X}$. Since V is finite and $T'|_{\mathrm{fin}}$ is finitary by definition, there exists a finite subset $\mathcal{L}' \subseteq \mathcal{L}$ and a function B' such that the following diagram commutes.

$$\begin{array}{ccccc} V & \xrightarrow{B} & T'|_{\mathrm{fin}}(\mathcal{L}_\epsilon \times V + Y) & \xrightarrow{\cong} & T'|_{\mathrm{fin}}(V + (\mathcal{L} \times V + Y)) \\ & \searrow^{B'} & & & \uparrow \\ & & & & T'|_{\mathrm{fin}}(V + (\mathcal{L}' \times V + Y)) \end{array}$$

Now $V+(\mathcal{L}'\times V+Y)$ is a finite set; hence, $T'|_{\mathrm{fin}}(V+(\mathcal{L}'\times V+Y))$ $= T'(V+(\mathcal{L}'\times V+Y))$. We can therefore apply the iteration operator $iter$ to B'; the result is

$$V \xrightarrow{iter(B')} T'(\mathcal{L}' \times V + Y) = T'|_{\mathrm{fin}}(\mathcal{L}' \times V + Y)$$

since $\mathcal{L}'\times V+Y$ is a finite set. Let us define B'' as the composition of

$$V \xrightarrow{iter(B')} T'|_{\mathrm{fin}}(\mathcal{L}' \times V + Y) \hookrightarrow T'|_{\mathrm{fin}}(\mathcal{L}_\epsilon \times V + Y)$$

and define $\varepsilon\text{-elim}_{\mathrm{fin}}(G) \stackrel{\mathrm{def}}{=} (V, B'', I)$ in $T'|_{\mathrm{fin}}$-$\mathcal{G}_{\mathrm{f}Y}^{\ X}$.

Then, for a finite $T'|_{\mathrm{fin}}$-graph G, by the inclusion $T'|_{\mathrm{fin}} \hookrightarrow T'$, we can regard G also as a T'-graph. It can be easily checked that $\varepsilon\text{-elim}(G)$ in Definition 5 is exactly equal to the above $\varepsilon\text{-elim}_{\mathrm{fin}}(G)$ regarded as a T'-graph.

For a finitary monad T with an extension for ε-elimination T', the condition that T is finitary can be used to show that $\iota\colon T \to T'$ can be decomposed into the inclusion $T'|_{\mathrm{fin}} \subseteq T'$ and some (necessarily injective and unique) monad morphism $\iota_{\mathrm{fin}}\colon T \to T'|_{\mathrm{fin}}$. Then $(T, T'|_{\mathrm{fin}}, \iota_{\mathrm{fin}}, \varepsilon\text{-elim}_{\mathrm{fin}})$ becomes a "finitary monad extension for ε-elimination" which is almost a monad extension for ε-elimination except for that $\varepsilon\text{-elim}_{\mathrm{fin}}$ can perform ε-elimination only for finite graphs.

B. STRONG-BISIMULATION GENERICITY OF STRUCTURAL RECURSION

We give a proof of Lemma 16—the strong-bisimulation genericity of the structural recursion.

We will use the following notions and notations in the proof of the lemma. We write $R : A_1 \nrightarrow A_2$ for a relation $R \subseteq A_1 \times A_2$; e.g., the diagram on the left below means that, if $a_1\, R\, a_2$, then $f_1(a_1)\, S\, f_2(a_2)$.

$$\begin{array}{ccc} A_1 & \xrightarrow{f_1} & B_1 \\ {\scriptstyle R}\!\!+ & & +\!\!{\scriptstyle S} \\ A_2 & \xrightarrow[f_2]{} & B_2 \end{array} \qquad \begin{array}{ccccc} X_1 & \xrightarrow{I_1} & V_1 & \xrightarrow{B_1} & T(\mathcal{L}_\epsilon \times V_1 + Y_1) \\ {\scriptstyle S}\!\!+ & & {\scriptstyle R}\!\!+ & & +\!\!{\scriptstyle T(\mathcal{L}_\epsilon \times R + T)} \\ X_2 & \xrightarrow[I_2]{} & V_2 & \xrightarrow[B_2]{} & T(\mathcal{L}_\epsilon \times V_2 + Y_2) \end{array}$$

For $S : X_1 \nrightarrow X_2$ and $T : Y_1 \nrightarrow Y_2$, let us define the relation $\sim_{\mathrm{s}}{}^{S}_{T}\colon \mathcal{G}_{\mathrm{f}Y_1}^{\ X_1} \nrightarrow \mathcal{G}_{\mathrm{f}Y_2}^{\ X_2}$: for $G_i \in \mathcal{G}_{\mathrm{f}Y_i}^{\ X_i}$, $G_1 \sim_{\mathrm{s}}{}^{S}_{T} G_2$ if there exists a relation $R : V_1 \nrightarrow V_2$ such that the right diagram above commutes (i.e., each of the two squares holds). We write simply X for the diagonal relation between X and X, then note that $\sim_{\mathrm{s}}{}^{X}_{Y}\colon \mathcal{G}_{\mathrm{f}Y}^{\ X} \nrightarrow \mathcal{G}_{\mathrm{f}Y}^{\ X}$ is the same as $\sim_{\mathrm{s}}$.

PROOF. Let G_i be (V_i, B_i, I_i). From $G_1 \sim_{\mathrm{s}} G_2$, there exists $R : V_1 \nrightarrow V_2$ such that the following diagram commutes.

$$\begin{array}{ccccc} X & \xrightarrow{I_1} & V_1 & \xrightarrow{B_1} & T(\mathcal{L}_\epsilon \times V_1 + Y) \\ \| & & {\scriptstyle R}\!\!+ & & +\!\!{\scriptstyle T(\mathcal{L}_\epsilon \times R + Y)} \\ X & \xrightarrow[I_2]{} & V_2 & \xrightarrow[B_2]{} & T(\mathcal{L}_\epsilon \times V_2 + Y) \end{array} \tag{5}$$

Next, from the assumption that $e_1 \sim_{\mathrm{s}} e_2$, the following diagram commutes.

$$\begin{array}{ccccccc} & & & \overset{T(\langle \widetilde{e_1}, \pi_r \rangle + id_Y)}{} & & \overset{t \circ T([p,q])}{} & \\ V_1 & \xrightarrow{B_1} & T(\mathcal{L}_\epsilon \times V_1 + Y) & \to & T(\mathcal{G}_{\mathrm{f}Z}^{\ Z} \times V_1 + Y) & \to & \mathcal{G}_{\mathrm{f}Z \times V_1 + Z \times Y}^{\ Z} \\ {\scriptstyle R}\!\!+ & & {\scriptstyle T(\mathcal{L}_\epsilon \times R + Y)}\!\!+ & & {\scriptstyle T(\sim_{\mathrm{s}}{}^{Z}_{Z} \times R + Y)}\!\!+ & & +\!\!{\scriptstyle \sim_{\mathrm{s}}{}^{Z}_{Z \times R + Z \times Y}} \\ V_2 & \xrightarrow[B_2]{} & T(\mathcal{L}_\epsilon \times V_2 + Y) & \to & T(\mathcal{G}_{\mathrm{f}Z}^{\ Z} \times V_2 + Y) & \to & \mathcal{G}_{\mathrm{f}Z \times V_2 + Z \times Y}^{\ Z} \\ & & & \underset{T(\langle \widetilde{e_2}, \pi_r \rangle + id_Y)}{} & & \underset{t \circ T([p,q])}{} & \end{array}$$

Then, for the composition of the above, the transposition of V_i by $oplus$ yields

$$\begin{array}{ccc} 1 & \xrightarrow{G_1'} & \mathcal{G}_{\mathrm{f}Z \times V_1 + Z \times Y}^{\ Z \times V_1} \\ \| & & +\!\!{\scriptstyle \sim_{\mathrm{s}}{}^{Z \times R}_{Z \times R + Z \times Y}} \\ 1 & \xrightarrow[G_2']{} & \mathcal{G}_{\mathrm{f}Z \times V_2 + Z \times Y}^{\ Z \times V_2} \end{array} \quad \text{i.e., } G_1' \sim_{\mathrm{s}}{}^{Z \times R}_{Z \times R + Z \times Y} G_2'. \tag{6}$$

After that, we can show—see the remark below—that

$$\mathbf{cycle}(G_1') \sim_{\mathrm{s}}{}^{Z \times R}_{Z \times Y} \mathbf{cycle}(G_2'), \tag{7}$$

and from the left square of the diagram (5) we get

$$\lfloor id_Z \times I_1 \rfloor \mathbin{@} \mathbf{cycle}(G_1') \sim_{\mathrm{s}}{}^{Z \times X}_{Z \times Y} \lfloor id_Z \times I_2 \rfloor \mathbin{@} \mathbf{cycle}(G_2')$$

i.e., $\mathbf{srec}(e_1)(G_1) \sim_{\mathrm{s}} \mathbf{srec}(e_2)(G_2)$. □

We give a remark on Equation (7). As seen above, $T^{\mathcal{G}}(Y) = \mathcal{G}_{\mathrm{f}Y}$ (and also $\mathcal{G}_Y$) becomes a monad; then, $\mathbf{cycle}$ becomes an iteration operator in the Kleisli category. Furthermore, the iteration operator is uniform on morphisms that come from the Kleisli category $\mathbf{Set}_T$ via γ, if the iteration operator for T is uniform. The above derivation of Equation (7) from Equation (6) is just the *uniformity principle* on relations [12]. (Since we need the uniformity of $\mathbf{cycle}$ only on functions, so we need the uniformity of $iter^T$ only on functions as well.)