ISSN 1884-0760

GRACE TECHNICAL REPORTS

Parameterized Graph Transformation Languages with Monads

Kazuyuki Asada   Soichiro Hidaka   Hiroyuki Kato
Zhenjiang Hu   Keisuke Nakano

GRACE-TR-2012-07   October 2012

CENTER FOR GLOBAL RESEARCH IN
ADVANCED SOFTWARE SCIENCE AND ENGINEERING
NATIONAL INSTITUTE OF INFORMATICS
2-1-2 HITOTSUBASHI, CHIYODA-KU, TOKYO, JAPAN

WWW page: http://grace-center.jp/

Parameterized Graph Transformation Languages with Monads

Kazuyuki Asada Soichiro Hidaka
Hiroyuki Kato Zhenjiang Hu
National Institute of Informatics, Japan
{asada,hidaka,kato,hu}@nii.ac.jp

Keisuke Nakano
University of Electro-Communications, Japan
ksk@cs.uec.ac.jp

Abstract

UnCAL was introduced as a graph transformation language for unordered finite graphs, and has a powerful transformation method for finite graphs: structural recursion. Although UnCAL is powerful and recently applied to model driven software engineering, there are two major limitations. One is that its graph model is only unordered graph in which branches of nodes are unordered. The other is its limited expressive power: structural recursion functions transform depth-direction well, but not well on sibling-direction. We solve these problems by generalizing the graph model and extending the expressive power of UnCAL. We generalize the type of branches of nodes from powerset to general monad, so that we introduce graph transformation languages $\lambda_{\mathrm{FG}}^{T}$ which are parameterized by (finitary) monads T. The special case where T is the finite powerset monad becomes an extension of UnCAL. Our crucial instance is the case of list monad for treating ordered graph, where we solve how to define bisimilarity between ordered graphs having edges labeled by an invisible label ε. Also we extend the expressive power of $\lambda_{\mathrm{FG}}^{T}$ from that of UnCAL: higher order functions and transformations for sibling dimension. We demonstrate that we can modularize designing graph transformation languages, by our generalization with monads and by monad transformers.

1. Introduction

Structural recursion, such as fold on lists or catamorphism [Meijer et al. 1991] on algebraic data structures including trees, plays an important role in functional programming, providing a systematic way for construction and manipulation of functional programs [Bird and de Moor 1996; Gill et al. 1993; Hu et al. 2006]. It is, however, a challenge to define structural recursions for graph data structures, the most ubiquitous data in computing. This is because unlike lists and trees, graphs are essentially not inductive and cannot be formalized as an initial algebra in general [Gibbons 1995].

It is certainly possible to use full recursion operator (fixed point operator) to manipulate *infinite* data structures such as stream and graph, but there are many cases targeting only *finite* graphs especially in database application. Therefore, it is very much desirable that we could represent graphs in finite form, while guaranteeing transformations (or queries) for graphs always terminate.

UnCAL [Buneman et al. 2000] was introduced in the database community to provide such a powerful querying method by structural recursion for finite graphs. The graph model of UnCAL is unordered (directed) graph, where outgoing edges are not ordered. In UnCAL we can treat graphs semantically as *regular trees* [Ginali 1979], based on a suitable definition of bisimulation, up to which finite graphs and regular trees are equivalent.

The benefit from this bisimulation is that structural recursion on regular trees can be used for graphs, and moreover, it gives an interesting and important semantics of structural recursion on graphs: *bulk semantics*. With bulk semantics, a structural recursion is evaluated by first processing *in parallel* on all edges of the input graph and then combining the results. This bulk semantics relies on introduction of ε-edges (like ε transition in automata) to graphs, providing a smart way of treating shared nodes and cycles in graphs. In addition, UnCAL is designed carefully such that (i) every transformation is *bisimulation generic* in the sense that it returns bisimilar results for bisimilar inputs, and (ii) every transformation necessarily *terminates* and transforms *finite* graphs to *finite* graphs.

Despite its beauty in theory and usefulness in graph querying, there are two limitations in UnCAL that prevent it from being used widely. One is that UnCAL can treat only unordered graphs, being weak in other graph models, such as ordered graphs where edges from a node are ordered, which is widely used in many applications: e.g., in Ecore, MOF, and KM3 [Jouault and Bézivin 2006]. The other limitation is lack of expressive power of transformations on sibling dimension. For example, when a graph labeled with natural number, we can not write in UnCAL a transformation which extracts all such edges of some node that are labeled with the average number among all the siblings.

At the first sight, it seems that the first limitation could be easily solved by encoding ordered graphs by unordered ones with suitable edge labels, say using hd and tl to represent the first branch and the rest of the branches respectively. However, there is a fatal problem with this encoding. The above encoding is *unsound* in the context of UnCAL where ϵ-edges must be taken into account for graph construction and structural recursion. As shown in Figure 1, the ordered graphs G_1 and G_2, where we use dotted arrows to represent ε-edges, are naturally bisimilar, but their corresponding encoded graphs G_1' and G_2' are not at all.

In fact, it has been an *open problem* for more than ten years whether structural recursion in UnCAL can be extended from unordered graphs to ordered ones or not, since it was first raised in the conclusion of the paper [Buneman et al. 2000]:

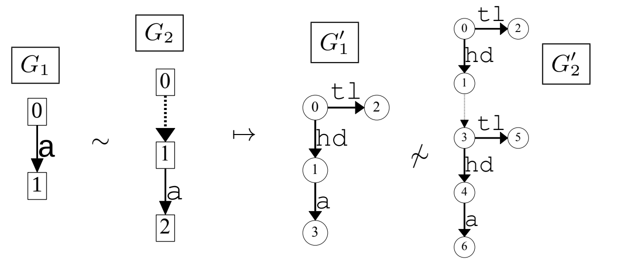


Figure 1. An Example of Unsound Encoding

... we have shown how the principles of UnQL will work on an ordered tree. However, it is not clear how they can be extended to an ordered graph model. ... we still lack a complete picture of this topic ...

In this paper, we provide a novel solution to this open problem, by defining a subtle notion of bisimilarity for ordered graph having ε-edges so that structural recursion can be extended from that for unordered graphs to that for ordered graphs. Moreover, we propose $\lambda_{\mathrm{FG}}^{T}$, powerful higher order graph transformation languages, which extends the lambda calculus with graph operations generalized with monads and also extended with sibling transformation.

The main technical contributions of this paper can be summarized as follows.

There are mainly two independent contributions; (i) one is the definition of bisimilarity for ordered graphs, which has a subtle problem of infinite width, which can not be seen in other graph models; (ii) the other one is the generalization with monads T and extension of expressive power of graph transformations: introduction of higher order function and sibling transformation. Further detail of these contributions are the following:

(i). We give the first definition of the bisimilarity between ordered graphs having ε-edges, which forms the semantic foundation for $\lambda_{\mathrm{FG}}^{List}$. Specifically, we identify that the branch order of even finite graphs is not necessarily finite but *countable linear order*, and clarify that combination of ε-edges and cycles will induce such countable linear order on branching, which would produce new issues not occurring for the unordered case. We show that the judgment of emptiness of ordered graphs is decidable, and that it is decidable whether we can eliminate all ε-edges—with keeping finite width of graph—for a finite ordered graph having ε-edges.

(ii). UnCAL consists of the following three technical points:

- graph models and bisimilarity,
- syntactic graph representation (called *graph constructors*), and
- structural recursion.

We generalize all the three points with monads T and extend further:

- We generalize the graph model in such a way that collection of outgoing edges of a node has the structure of the monad T. With this, unordered graph and ordered graph correspond to finite powerset monad and list monad, respectively. We can see its usefulness from other examples of multiset monad and distribution monad which correspond to weighted graph and probability graph, respectively.

  Then we give a definition of bisimilarity for such graph models generally with T. The above (i), i.e., the definition of bisimilarity for ordered graph is independent contribution from this general definition; since for the general definition we assume existence of *iteration operators*, which is newly given in this paper for ordered graph, while those for other graph models are already known.
- Then according to the generalization of graph model, we also give generalized definition of graph constructors and structural recursion with monads. Specifically, we generalize *bulk semantics* of the structural recursion, and show that structural recursion is *bisimulation generic*. The bisimulation genericity is a stronger result than that in [Buneman et al. 2000] even for the case of unordered graph, since here we prove bisimulation genericity of structural recursion as a higher order function, while in [Buneman et al. 2000] it is proved as first order functions. This higher order bisimulation genericity enables us to introduce higher order functions to UnCAL and reformulation as a lambda calculus.
- We define our graph transformation languages $\lambda_{\mathrm{FG}}^{T}$ as extensions of the simply typed lambda calculus, which is in sharp contrast to UnCAL (first order calculus). This reformulation does not only provides us higher order functions, but also a clear vision of how to extend $\lambda_{\mathrm{FG}}^{T}$ further, with other primitive functions or other type systems such as coproduct types, inductive data types, monad types, and polymorphic types.
- We show how to overcome the second limitation of UnCAL, i.e., expressive power on sibling dimension, by introducing two *sibling transformation* **l-sbl** and **u-sbl** upon the above flexibility of extensions. For instance, when $T = List$, we can manipulate branches of nodes as we transform lists: we can, for example, reverse the order of branches of a node in an ordered graph with reverse function of list.
- Summarizing the above, we formulate parametrized graph transformation languages $\lambda_{\mathrm{FG}}^{T}$ with monads T.

(iii). Then we have the following discussion for further extension.

- We extend the structural recursion to unify the above sibling transformations with the extended structural recursion, so that we can transform graphs in depth-direction and in sibling-direction at the same time.
- Finally we discuss how we can systematically compose graph transformation languages to increase language power, by using monad transformer. For instance, although $\lambda_{\mathrm{FG}}^{List}$ can treat ordered graphs but not unordered graphs, by this modular method, we can systematically obtain $\lambda_{\mathrm{FG}}^{P_{\mathrm{fin}} \times List}$, which can treat both unordered and ordered graphs.

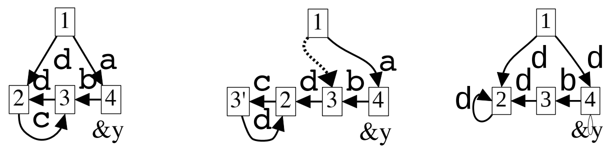
Organization of the Paper We shall start by an overview of $\lambda_{\mathrm{FG}}^{T}$, focusing on the case when $T = List$, in Section 2. Then in Section 3, we see the first key technical contribution of this paper, i.e., the definition of bisimilarity for ordered graphs having ε-edges. Then we give a general definition of bisimilarity for graphs in $\lambda_{\mathrm{FG}}^{T}$. In Section 4, we give interpretations of terms of $\lambda_{\mathrm{FG}}^{T}$, which includes graph constructors, structural recursion, and sibling transformations; then we show their bisimulation genericity. In Section 5, we extend structural recursion with sibling transformation. In Section 6, we discuss how we can extend the languages with monad transformers. We discuss the related work in Section 7 and conclude the paper in Section 8.

2. Overview of $\lambda_{\mathrm{FG}}^{T}$

We start with an overview of our general graph transformation language $\lambda_{\mathrm{FG}}^{T}$, especially when $T = List$. In fact, $\lambda_{\mathrm{FG}}^{List}$, which treats ordered graphs, is the most important case to understand essence of $\lambda_{\mathrm{FG}}^{T}$. In the following, after giving general definition of graphs, we will move to the case of $List$, and see how ordered graphs are constructed and how structural recursion can be easily used to transform ordered graphs. In Section 4.4, we will see how to design the syntax of general $\lambda_{\mathrm{FG}}^{T}$.

2.1 Graph Model of $\lambda_{\mathrm{FG}}^{T}$

Graph in $\lambda_{\mathrm{FG}}^{T}$ is rooted, directed, and edge-labeled graph. Additionally, graph in $\lambda_{\mathrm{FG}}^{T}$ has two prominent features, *ε-edges* and *markers*. An ε-edge represents a shortcut between the two nodes, working like the ε-transition in an automaton. Nodes may be marked with *input* and *output markers*, which are used as an interface to connect them to other graphs by ε-edges.



(a) A Simple Graph: G (b) An Equivalent Graph: G' (c) Result of $a2d\_xc$ on (a)

Figure 2. Examples of Graphs

We introduce the notion of graph with "T-kind of branch" for a monad T. First let us recall the notion of monad (in the Kleisli triple style): $T = (T, return, lift)$ is a *monad* on $\mathbf{Set}$ if $T\colon \mathbf{Set} \to \mathbf{Set}$ is a functor,

$$return\colon S \to T(S)$$

and

$$lift\colon (S \to T(S')) \to (T(S) \to T(S'))$$

are natural transformations, and they satisfy certain axioms called monad laws, see [Benton et al. 2000; Moggi 1989] for the axioms.

Example 1 (List Monad)**.** $List$ is a monad defined as follows.

$$return(s) = [s] \quad (s \in S)$$
$$lift\ f = concat \circ List(f) \quad (f\colon S \to List(S'))$$

where

$$concat\colon List(List(S')) \to List(S')$$

is flattening a list of lists to a list. □

Now we 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.

Note that in the terminology of coalgebra theory, a T-graph is a coalgebra B of the endofunctor $T(\mathcal{L}_\epsilon \times (\text{-}) + Y)$ equipped with $|X|$-number of initial states I.

Example 2 (Ordered Graph)**.** We call $List$-graphs *ordered graphs*, where the branches are ordered. The graph in Figure 2(a) is represented as (V, B, I) where

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

□

The set of graphs—with X and Y as sets of input and output markers, respectively—is denoted by $T\text{-}DB^X_Y$, where T may be omitted if it is clear from context. We call a T-graph a *finite* T-graph when V is a finite set, and write $T\text{-}DB_{\mathrm{f}}{}^X_Y$ for the set of finite T-graphs. ("DB" is from [Buneman et al. 2000] and means "DataBase".)

We occasionally use record notation (-).V, (-).B, and (-).I for components of a graph: i.e., $G = (G.\mathrm{V}, G.\mathrm{B}, G.\mathrm{I})$.

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

An output marker is a place holder through which the input node of another graph can be connected with an ε-edge. (This is done with @, $\mathbf{cycle}$ (see Figure 3), and $\mathbf{srec}$ (see Figure 7) to be explained later.)

Example 3 (Unordered Graph, Weighted Graph, Probability Graph)**.** For the finite powerset monad P_{fin}, P_{fin}-graph is *unordered graph*, which is (equivalent to) the graph model of UnCAL.

Let us consider the *finite multiset monad* (bag monad) M_{fin}:

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

Then branch of M_{fin}-graph has a bag semantics (rather than set semantics of P_{fin}), i.e., multiplicity (called *weight*) of an identical branch is not ignored.

The *finite probability distribution monad* D_{fin} is defined as:

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

The monad structure is defined as below: for $s \in S$, $return(s)$ is the Dirac delta function

$$\begin{array}{rl} \delta_s\colon S & \to [0,1] \\ s & \mapsto 1 \\ (\text{other than } s) & \mapsto 0 \end{array}$$

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

$$lift(f)\colon D_{\mathrm{fin}}(S) \to D_{\mathrm{fin}}(S')$$
$$(\phi\colon S \to [0,1]) \mapsto \begin{pmatrix} lift(f)(\phi)\colon S' \to [0,1] \\ \qquad s' \mapsto \Sigma_{s\in S}(\phi(s)\cdot f(s)(s')) \end{pmatrix}$$

D_{fin}-graph has probabilistic branch. □

2.2 Overview of $\lambda^{List}_{\mathrm{FG}}$

From now we see an overview of λ^T_{FG} when $T = List$.

2.2.1 Graph Equivalence

For ordered graphs, we consider bisimilarity as their graph equivalence, which is called *value-equivalence* in [Buneman et al. 2000], since we observe only "values" l and $\&y$ pointed by a "pointer" v. For instance, the graphs G and G', in Figure 2(a) and 2(b), respectively, are considered as being equivalent. In G', nodes 3 and 3$'$ are bisimilar because both of them only have one outgoing edge labeled d to the node 2. Also in G', from node 1 to node 3, there is an ε-edge (denoted by the dotted line), which can be eliminated with keeping its bisimilarity by adding outgoing edge labeled d from node 1 to node 2. Unreachable parts from roots are disregarded. The definition of bisimulation on ordered graphs with ε-edges is one of the important results in this paper, and this will be addressed formally in Section 3.

A graph function f is called *bisimulation generic* if $f(G)$ and $f(G')$ are bisimilar whenever G and G' are bisimilar. λ^T_{FG} is designed to make the interpretation of every term bisimulation generic.

2.2.2 Graph Constructors

Figure 3 summarizes the nine graph constructors used in $\lambda^{List}_{\mathrm{FG}}$, and their typing rules are shown in Figure 5. They are similar with graph constructors for unordered graphs in UnCAL [Buneman et al. 2000]. Note that these constructors should be written as e.g. $[]_Y$

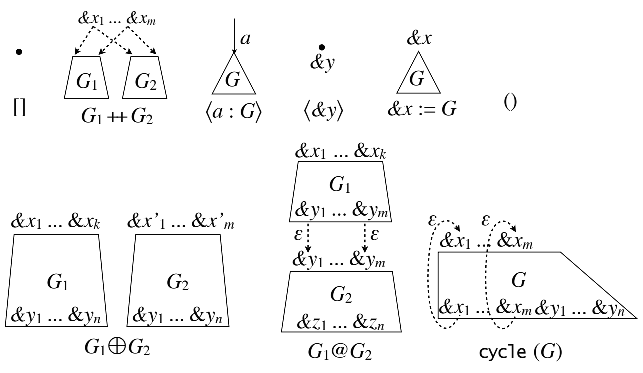


Figure 3. Graph Constructors

$$
\begin{array}{rll}
e ::= & x \mid \lambda x.e \mid ee \mid \textbf{case } e \textbf{ of in}^{\mathbf{l}}(x) \to e \textbf{ or in}^{\mathbf{r}}(y) \to e & \\
\mid & \textbf{in}^{\mathbf{l}}\, e \mid \textbf{in}^{\mathbf{r}}\, e \mid (e, e) \mid \pi^{\mathbf{l}} e \mid \pi^{\mathbf{r}} e & \{ \text{ terms of lambda calculus } \} \\
\mid & \textbf{if } e \textbf{ then } e \textbf{ else } e & \{ \text{ conditional } \} \\
\mid & a \mid \&y \mid e = e & \{ \text{ label } (a \in \mathcal{L}), \text{ marker, and their equality } \} \\
\mid & [\,] \mid e +\!\!\!+ e & \{ \text{ algebraic graph constructors } \} \\
\mid & \langle e : e \rangle \mid \langle \&y \rangle \mid \&x := e \mid () \mid e \oplus e \mid e \mathbin{@} e \mid \textbf{cycle}(e) & \\
 & & \{ \text{ common graph constructors } \} \\
\mid & \textbf{isEmpty}(e) & \{ \text{ graph emptiness checking } \} \\
\mid & \textbf{srec}(e)(e) & \{ \text{ structural recursion application } \} \\
\mid & \textbf{nil} \mid \textbf{cons}(e, e) \mid \textbf{foldr}(e, e, e) \mid \ldots & \{ \text{ list operators } \} \\
\mid & \textbf{l-sbl}(e)(e) \mid \textbf{u-sbl}(e)(e) & \{ \text{ sibling transformations } \}
\end{array}
$$

Figure 4. $\lambda_{\mathrm{FG}}^{List}$ Language

and $G_1 +\!\!\!+_{X,Y}\, G_2$; however we will omit the subscript X and Y to avoid clutter.

Let us see each constructor; also see how type discipline on input and output markers works for each constructor. First, [] constructs a root-only graph with the default input marker and no output markers. For two graphs G_1 and G_2 having identical input markers and output markers, $G_1 +\!\!\!+ G_2$ adds two branching ε-edges from each new root to the corresponding old roots of G_1 and G_2. Next, $\langle a : G\rangle$ extends G with one a-labeled edge pointing the old root from a new fresh root node; and the constructor $\langle \&y \rangle$ constructs a graph with a single node marked with an output marker $\&y$ (in [Buneman et al. 2000], $\langle a : G\rangle$ and $\langle \&y \rangle$ are denoted as $\{a : G\}$ and $\&y$, respectively). *Marker renaming* $\&x := G$ associates an input marker $\&x$ to the root node of G; () constructs a trivial graph that has neither a node nor an edge; and $G_1 \oplus G_2$ constructs a disjoint union of G_1 and G_2, where their branching functions B_1 and B_2 work independently. Then, $G_1 \mathbin{@} G_2$ composes two graphs sequentially by connecting the output nodes of G_1 with the corresponding input nodes of G_2 by ε-edges, and $\textbf{cycle}(G)$ connects the output nodes with the input nodes of G to form cycles. It is worth noting that this set of constructors are powerful enough to describe any finite ordered graphs.

2.2.3 Syntax of $\lambda_{\mathrm{FG}}^{List}$

Our graph transformation language $\lambda_{\mathrm{FG}}^{List}$ (and in general $\lambda_{\mathrm{FG}}^{T}$) is an extension of the simply typed lambda calculus with graph constructors and graph operations. The syntax of $\lambda_{\mathrm{FG}}^{List}$ is depicted in Figure 4, and the types and typing rules are in Figure 5.

The type $\mathbf{DB}_Y^X$ is interpreted to $DB\mathrm{f}_Y^X$: the set of *finite* graphs, and α are type variables, which are used just for **u-sbl** and explained later. We omit the standard explanations for lambda terms, conditional, label, and equality for labels.

$$
\begin{array}{rll}
\sigma ::= & \sigma + \sigma \mid \sigma \times \sigma \mid \sigma \to \sigma & \{ \text{ variant, product, function types } \} \\
\mid & \mathbf{Bool} \mid \mathbf{Label} \mid \mathbf{DB}_Y^X & \{ \text{ base types } \} \\
\mid & \alpha \mid Y \mid \mathbf{List}(\sigma) & \{ \text{ types for sibling transformation } \}
\end{array}
$$

$$
\frac{}{\Gamma \vdash [\,] : \mathbf{DB}_\emptyset}
\qquad
\frac{\begin{array}{c}\Gamma \vdash e_1 : \mathbf{DB}_Y^X \\ \Gamma \vdash e_2 : \mathbf{DB}_Y^X\end{array}}{\Gamma \vdash e_1 +\!\!\!+ e_2 : \mathbf{DB}_Y^X}
\qquad
\frac{\begin{array}{c}\Gamma \vdash e_1 : \mathbf{Label} \\ \Gamma \vdash e_2 : \mathbf{DB}_Y\end{array}}{\Gamma \vdash \langle e_1 : e_2 \rangle : \mathbf{DB}_Y}
$$

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

$$
\frac{\begin{array}{c}\Gamma \vdash e_1 : \mathbf{DB}_Y^{X_1} \;\; \Gamma \vdash e_2 : \mathbf{DB}_Y^{X_2} \\ (X_1 \cap X_2 = \emptyset)\end{array}}{\Gamma \vdash e_1 \oplus e_2 : \mathbf{DB}_Y^{X_1 \cup X_2}}
\qquad
\frac{\begin{array}{c}\Gamma \vdash e_1 : \mathbf{DB}_Y^X \\ \Gamma \vdash e_2 : \mathbf{DB}_Z^Y\end{array}}{\Gamma \vdash e_1 \mathbin{@} e_2 : \mathbf{DB}_Z^X}
$$

$$
\frac{\Gamma \vdash e : \mathbf{DB}_{X \cup Y}^X \quad (X \cap Y = \emptyset)}{\Gamma \vdash \mathbf{cycle}(e) : \mathbf{DB}_Y^X}
\qquad
\frac{\Gamma \vdash e : \mathbf{DB}_Y^X}{\Gamma \vdash \mathbf{isEmpty}(e) : \mathbf{Bool}}
$$

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

$$
\frac{\begin{array}{c}\Gamma,\ x{:}\mathbf{List}(\mathbf{Label} \times \mathbf{DB}_Y + Y) \vdash e_1 : \mathbf{List}(\mathbf{Label} \times \mathbf{DB}_Y + Y) \\ \Gamma \vdash e_2 : \mathbf{DB}_Y\end{array}}{\Gamma \vdash \mathbf{l\text{-}sbl}(\lambda x.e_1)(e_2) : \mathbf{DB}_Y}
$$

$$
\frac{\begin{array}{c}\Gamma,\ x{:}\mathbf{List}(\mathbf{Label} \times \alpha + Y) \vdash e_1 : \mathbf{List}(\mathbf{Label} \times \alpha + Y) \\ \Gamma \vdash e_2 : \mathbf{DB}_Y^X\end{array}}{\Gamma \vdash \mathbf{u\text{-}sbl}(\lambda x.e_1)(e_2) : \mathbf{DB}_Y^X}
\qquad
\frac{}{\Gamma \vdash \&y : Y}
$$

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

Figure 5. Types and Typing Rules of UnCAL^{List}

The graph constructors here are described separately, divided into *algebraic graph operators* and *common graph constructors*. Algebraic graph operators depend on T, while the seven common graph constructors are fixed independently from T. In fact, algebraic graph operators further depend on *signatures* of T, as seen in Section 6; but independently from choice of such signatures, their expressive power remains the same not just on closed terms but also on open terms. The boolean expression $\textbf{isEmpty}(e)$ results in $true$ when the graph of the result of e has no non-ε edge in the accessible part.

Structural Recursion Now we explain *structural recursion*, which is a powerful mechanism borrowed from UnCAL [Buneman et al. 2000] to describe graph transformations. The structural recursion $f = \textbf{srec}(\lambda(l, g).e)$ is such a function that f satisfies the following equations, where we ignore output markers and consider the case when X is singleton, for simplicity:

$$
\begin{array}{lcl}
f([\,]) & = & [\,] \\
f(\langle l : g \rangle) & = & e(l, g) \mathbin{@} f(g) \\
f(g_1 +\!\!\!+ g_2) & = & f(g_1) +\!\!\!+ f(g_2)
\end{array}
$$

The above equations give a definition of f as a function which inputs *finite* graphs and outputs *infinite* graphs, in recursive way. However the outputs of f are in fact (bisimilar to) *finite* graphs, and it is made clear by the *bulk semantics* of f, which will be given in Section 4.2. Intuitively, in the bulk semantics, structural recursion f transforms a graph in a bulk way by keeping structure of the graph—as map functions for list or tree—, transforming each edge labeled l to new graph parts $e(l, g)$. Here e can refer not only the label l but also its successor graph g in the input graph of f.

Example 4. The following structural recursion $a2d\_xc$ replaces all labels `a` with `d` and contracts edges labeled `c`.

$$
\begin{aligned}
&a2d\_xc = \mathbf{srec}(rc)\\
&\quad \mathbf{where}\\
&\qquad rc = \lambda(l,g).\,\mathbf{if}\ \ l = \mathtt{a}\ \mathbf{then}\ \langle \mathtt{d} : \langle \& \rangle\rangle\\
&\qquad\qquad \mathbf{else\ if}\ \ l = \mathtt{c}\ \mathbf{then}\ \langle \& \rangle\\
&\qquad\qquad\qquad \mathbf{else}\ \langle l : \langle \& \rangle\rangle
\end{aligned}
$$

Applying the function $a2d\_xc$ to the graph in Fig. 2(a) yields the graph in Fig. 2(c). □

Example 5. 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 [Robertson et al. 2009], computes the table of contents of books in which sections can be arbitrarily nested:

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

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

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

□

Sibling Transformation Structural recursion functions are powerful transformations which terminate at evaluation and preserve finiteness of graphs, but we need more expressive power: transformations on sibling dimension.

Let us consider an ordered graph and its unfolded (maybe infinite) tree. The branches under the root are then a list of the subtrees. Then if one wants to, e.g., reverse the order of the branches under the root, we can not do so with the structural recursion.

To resolve this problem, we introduce *local sibling transformations* **l-sbl** and *uniform sibling transformations* **u-sbl**, which enable such transformations. The former **l-sbl** transforms branches only of the root node of a single-rooted graph; and by combination with the structural recursion, **l-sbl** can transform branches of any one node which the structural recursion function can reach. The latter **u-sbl** transforms branches uniformly of all nodes in a graph.

Let us return to the Figure 4. List operators in the syntax are usual ones; we can add anything that is convenient such as **car**, **cdr**, **filter**, and **reverse** etc. These list operators are used for the two sibling transformations; $\mathbf{l\text{-}sbl}(\lambda x.e)$ and $\mathbf{u\text{-}sbl}(\lambda x.e)$ transform sibling as $\lambda x.e$ transforms lists. In the typing rule for **u-sbl**, we use type variables α to prepare a parametric polymorphic function $\lambda x.e$ on $\mathbf{List}(\mathbf{Label}{\times}\alpha{+}Y)$ so that $\lambda x.e$ and hence $\mathbf{u\text{-}sbl}(\lambda x.e)$ become "uniform" on α. The parametricity guarantees bisimulation genericity of **u-sbl**. For the detail see Section 4.3.

3. Bisimilarity for ε-edge and Ordered Graph

Section 2.1 gives a general definition of T-graph, for a monad T. To this end, we shall give the semantic equivalence for the graph model of λ^T_{FG}: bisimilarity between T-graphs.

The main point is a treatment of ε-edge. The case of ordered graph ($List$-graph) involves a big problem which does not occur for unordered graph (P_{fin}-graph): i.e., ε-elimination might induce *infinite* width. In the following, we shall see the problem and how to define bisimilarity between ordered graphs. Also we give some effective procedure to avoid such infinite width. Then we give a general definition of bisimilarity for T-graphs. Finally, we extend the bisimilarity equivalence from graph types to higher order function types.

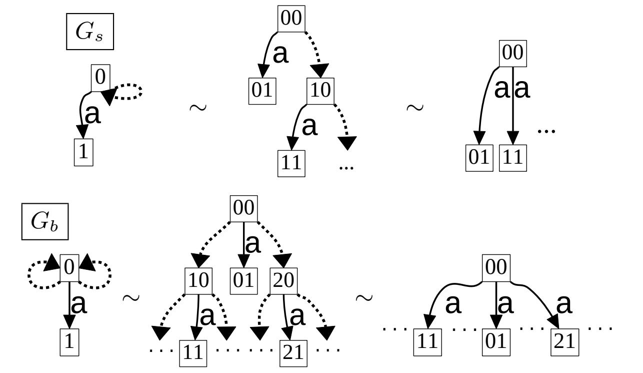


Figure 6. Graphs with Stream Branching and with Dense Branching

We here remark that our bisimilarity for the invisible label ε is different from *weak bisimilarity* for the invisible label τ in the context of process algebra [Milner 1999]. Our bisimilarity is characterized by the ε-elimination (Proposition 8.2) which is familiar in automata theory. One purpose of our use of ε-edge is to postpone calculation of graph constructors, structural recursion and so on, but weak bisimilarity is unsuitable with respect to properties of such graph transformations: e.g. associativity of $+\!\!+$ fails with weak bisimilarity.

3.1 Bisimilarity for Ordered Graph

Now we see the bisimilarity between ordered graphs: its intuition and formal definition.

Intuition of the Bisimilarity

First let us see some examples of ordered graphs in order to have some feeling of the bisimilarity to be defined. Consider the graphs in Figure 6. For the graph G_s, first, in order to make our problem easily understandable, let us unfold the graph to an infinite tree, i.e., the graph in the middle. Then intuitive "ε-elimination" of the tree is the graph on the right, where the branching is as a stream. Note that for ordered graph there is no *idempotency* which unordered graph has. The way of branching of graphs having ε-edges are thus possibly infinite essentially. However, it is not necessarily just a stream type, as in the next example G_b. Unfolding of graph G_b yields the tree in the middle, and then its "ε-elimination" is the graph on the right. This graph has a branch like the ordered set $\{n/2^m \in \mathbb{Q} \mid n,m \in \mathbb{N}, 0{<}n{<}2^m\}$, which is a dense countable linear ordered set. (In this paper, the term *countable* includes the case of finite.)

In fact, for any countable linear ordered set, there is some ordered graph having ε-edges such that the branches of the root after eliminating ε-edges are exactly as the given ordered set. (For the detail, see Appendix A.)

Formal Definition of the Bisimilarity

Now let us define bisimilarity between ordered graphs. As seen above, the ε-elimination of an ordered graph might induce countable width. So we shall first define a generalized notion of "ordered graph with countable width", and define bisimilarity for such generalized graphs. This asks us to extend the list monad

$$List(S) \overset{\mathrm{def}}{=} \Sigma_{n\in\mathbb{N}} S^n$$

to the countable list monad $CList$ which can be defined as the following:

$$CList(S) \overset{\text{def}}{=} \Sigma_{L\in\mathbb{L}} S^L$$

where $\mathbb{N}$ is generalized to $\mathbb{L}$, the set of countable linear ordered sets up to order isomorphism (with chosen representatives L). (See Appendix B for the precise definition of $CList$.) Thus we extend ordered graphs to $CList$-graphs.

For $B(v) \in \Sigma_{L\in\mathbb{L}}(\mathcal{L}_\epsilon \times V + Y)^L$, let $|B(v)|$ denote the countable linear ordered set L of $B(v)$. Then, we call $i \in |B(v)|$ *a branch index* of a node v, and write $B(v).i \in \mathcal{L}_\epsilon \times V + Y$ for the i-th branch.

Next we prepare the notion of "transitive closure" of ε-edges. Let $G = (V, B, I) \in CList\text{-}DB_Y^X$ and $v \in V$. Let us consider a pair of an ε-path from v and a branch index i_n of the last node in the ε-path:

$$v\ (= v_0) \overset{\varepsilon}{\to}_{i_0} v_1 \ldots \overset{\varepsilon}{\to}_{i_{n-1}} v_n \to_{i_n} \ (n \in \mathbb{N})$$

where $v_n \to_{i_n}$ means just that $i_n \in |B(v_n)|$. We call this *proper branch of* v if the i_n-th branch $B(v_n).i_n$ is not an ε-edge, i.e., it is either a non-ε edge or an output marker. The set of all proper branches of v in G is denoted by $\mathrm{Pb}(G, v)$. It is easily shown that $\mathrm{Pb}(G, v)$ is a countable set.

Then there is a natural linear order $\leq_{\mathrm{Pb}}$ on $\mathrm{Pb}(G, v)$: for two different proper branches

$$p = (v \overset{\varepsilon}{\to}_{i_0} v_1 \ldots \overset{\varepsilon}{\to}_{i_{n-1}} v_n \to_{i_n}),$$
$$p' = (v \overset{\varepsilon}{\to}_{i'_0} v'_1 \ldots \overset{\varepsilon}{\to}_{i'_{n'-1}} v'_{n'} \to_{i'_{n'}}),$$

we obtain their branch indices sequences: $\tilde{p} \overset{\text{def}}{=} (i_0, \ldots, i_{n-1}, i_n)$ and $\tilde{p'} \overset{\text{def}}{=} (i'_0, \ldots, i'_{n'-1}, i'_{n'})$. With the starting node v, we can recover p and p' from these indices sequences. Then, between $\tilde{p}$ and $\tilde{p'}$, we can consider lexicographical order $\leq_l$, then we define $p \leq_{\mathrm{Pb}} p' \overset{\text{def}}{\iff} \tilde{p} \leq_l \tilde{p'}$.

Now we define the bisimilarity.

Definition 6 (Bisimilarity)**.** For $CList$-graphs $G = (V, B, I), G' = (V', B', I') \in CList\text{-}DB_Y^X$, a relation R between V and V' is called a *bisimulation relation* if for any vRv', there is an order isomorphism $f\colon (\mathrm{Pb}(G, v), \leq_{\mathrm{Pb}}) \to (\mathrm{Pb}(G', v'), \leq_{\mathrm{Pb}})$ satisfying the following property. For any proper branch

$$p = (v \overset{\varepsilon}{\to}_{i_0} \ldots v_n \to_{i_n}) \in \mathrm{Pb}(G, v)$$

with

$$f(p) = (v' \overset{\varepsilon}{\to}_{i'_0} \ldots v'_{n'} \to_{i'_{n'}}) \in \mathrm{Pb}(G', v'),$$

- if $B(v_n).i_n = \mathsf{Edge}\,(l, u)$ for some $l \in \mathcal{L}, u \in V$, then there exists $u' \in V'$ such that $B'(v'_{n'}).i'_{n'} = \mathsf{Edge}\,(l, u')$ and uRu',
- if $B(v_n).i_n = \mathsf{Outm}\,(\&y)$ for some $\&y \in Y$, then $B'(v'_{n'}).i'_{n'} = \mathsf{Outm}\,(\&y)$.

Two graphs G and G' are *bisimilar* (denoted by $G \sim G'$) if there is a bisimulation relation R such that for every input marker $\&x \in X$, $I(\&x)\, R\, I'(\&x)$. □

Note that the bisimulation relation is an equivalence relation on $CList\text{-}DB_Y^X$. Next, let us define ε-elimination.

Definition 7 (ε-elimination)**.** For a $CList$-graphs $G = (V, B, I) \in CList\text{-}DB_Y^X$, *$\varepsilon$-elimination* $\varepsilon\text{-elim}(G)$ of G is a $CList$-graph $(V, B', I) \in CList\text{-}DB_Y^X$ where $|B'(v)| \overset{\text{def}}{=} \mathrm{Pb}(G, v)$ and for $p = (v \overset{\varepsilon}{\to}_{i_0} \ldots v_n \to_{i_n})$ in $|B'(v)|$, $B'(v).p \overset{\text{def}}{=} B(v_n).i_n$. □

Note that the ε-elimination does not change sets of nodes.

Proposition 8. *1. For all $CList$-graphs $G \in CList\text{-}DB_Y^X$, $\varepsilon\text{-elim}(G)$ has no ε-edge, and G and $\varepsilon\text{-elim}(G)$ are bisimilar.*

2. For $CList$-graphs $G, G' \in CList\text{-}DB_Y^X$, G and G' are bisimilar if and only if $\varepsilon\text{-elim}(G)$ and $\varepsilon\text{-elim}(G')$ are bisimilar. □

Here we first gave the definition of bisimilarity for graphs having ε-edges and then gave that of ε-elimination and the above proposition. However, we can define the bisimilarity for graphs having ε-edges by the equivalence stated in Proposition 8.2 with the ε-elimination and with the bisimilarity for graphs having no ε-edge; the latter can be easily derived from general coalgebraic definition of bisimilarity [Rutten 2000]. Then we see that such definition is not just equivalent to but almost the same as that in Definition 6. Thus we find that the key here is the definition of ε-elimination. In the following we give a general definition of bisimilarity for T-graphs by ε-elimination.

3.2 Decidability for Finite Width Graphs

Before going to the general setting with monads T, we give one important decidability result for $CList$-graphs.

Let $\mathrm{FG}\not\varepsilon$ be the set of finite $List$-graphs which are bisimilar to some finite $List$-graphs having no ε-edge. There is a procedure which answers, for a finite $List$-graph G, if G is in $\mathrm{FG}\not\varepsilon$ or not. If G is in $\mathrm{FG}\not\varepsilon$, we can effectively eliminate ε-edges; otherwise, it is impossible to eliminate ε-edges, keeping finite width. This procedure is as the following. First note that for each node accessible from a root, we can check if there is an ε-cycle—a cyclic path consisting only of ε-edges—on the node. Then if, for every accessible node with ε-cycle, there is no proper branch, then the input graph is in $\mathrm{FG}\not\varepsilon$; otherwise, not in $\mathrm{FG}\not\varepsilon$.

This is enough for run-time use of the query language $\lambda_{\mathrm{FG}}^{List}$, because, though we need ε-edge for implementation—for structural recursion and for efficiency of graph calculation—practical graphs in the real world has no ε-edge. If a user of the language writes such a practical query, then the result should return a graph in $\mathrm{FG}\not\varepsilon$; if it is an incorrect query not intended by the user, and then if the result has inevitable ε-edges, the procedure above can check it and can warn the user.

Note that, in the class $\mathrm{FG}\not\varepsilon$, since we can eliminate ε-edges, we can obtain familiar effective procedures for bisimilarity-checking and for obtaining the minimum graphs in a similar way to unordered graph. Also note that if G is not in $\mathrm{FG}\not\varepsilon$, then G is not empty, hence $\mathbf{isEmpty}$ is decidable.

3.3 Generic Definition of Bisimilarity

Now we define the notion of bisimilarity in general for T-graphs. We define this bisimilarity for any monad T whose Kleisli category $\mathbf{Set}_T$ has a uniform iteration operator.

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

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

and the composition $g \mathbin{\underline{\circ}} f$ of $f\colon S \to T(S')$ and $g\colon S' \to T(S'')$ is defined as

$$g \mathbin{\underline{\circ}} f \overset{\text{def}}{=} lift(g) \circ f\colon S \to T(S'').$$

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

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

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 iteration operator [Haghverdi 2000; Kakutani 2002], which is the dual notion of fixed point operator [Simpson and Plotkin 2000]; also, iteration operator is to while operator as coproduct type is to the boolean type. Though iteration operator can be defined for any category with finite coproducts, we here define directly on the Kleisli category $\mathbf{Set}_T$ of a monad T on $\mathbf{Set}$. An *iteration operator* iter on $\mathbf{Set}_T$ is a function which maps a function

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

to

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

such that the mapping satisfies the following axioms:

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

Further, iter is called *uniform* if for functions $f \colon S \to T(S')$, $g \colon S' \to T(S'{+}A)$ and $h \colon S \to T(S{+}A)$,

$$\mathit{iter}(g) \mathbin{\underline{\circ}} f = \mathit{iter}(h) \colon S \to T(A)$$

whenever

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

The axiom of uniformity is used for logical relation for iteration operator [Hasegawa 2002]. We use uniformity to show that strong bisimilarity implies bisimilarity.

The following characterization of ε-elimination as iteration operator is due to [Hasuo 2011; Jacobs 2010a].

Definition 9 (ε-elimination)**.** Let T be a monad and iter be an iteration operator in $\mathbf{Set}_T$. For an T-graph $G = (V, B, I) \in T\text{-}\mathit{DB}^X_Y$, its *$\varepsilon$-elimination* $\varepsilon\text{-elim}(G) \in T\text{-}\mathit{DB}^X_Y$ is (V, B', I) where $B' \overset{\text{def}}{=} \mathit{embed} \circ \mathit{iter}(\mathit{iso} \circ B)$; here 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 when $\mathcal{L} = 0$. Then $B \colon V \to T(\{\varepsilon\}{\times}V{+}Y)$, and if we apply ε-elimination, the resulting branch function is $B' \colon V \to T(0{\times}V{+}Y)$. That is, we get an operator which maps a function $V \to T(V{+}Y)$ to a function $V \to T(Y)$. This is just the same as the structure of an iteration operator in the Kleisli category (if we allow Y to be arbitrary sets); and we find that it is natural to adopt the axioms of iteration operator also as axioms of ε-elimination.

Now we give a general definition of bisimilarity for T-graphs, using the above ε-elimination.

Before that, let us recall the notion of bisimulation relation for general endofunctor F on $\mathbf{Set}$. First we recall *F-lift* of relations: for (the inclusion function of) a relation $r \colon R \hookrightarrow V{\times}V'$, let $r_1 \overset{\text{def}}{=} \mathit{pr}_l \circ R \colon R \to V$ and $r_2 \overset{\text{def}}{=} \mathit{pr}_r \circ R \colon R \to V'$; so

$$\langle F(r_1), F(r_2) \rangle : F(R) \to F(V){\times}F(V').$$

Then $\tilde{F}(R)$ is defined as the image

$$\langle F(r_1), F(r_2) \rangle \, (F(R)) \subseteq F(V){\times}F(V').$$

For a functor F on $\mathbf{Set}$, and 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 $(B{\times}B')(R) \subseteq \tilde{F}(R)$.

Definition 10 (Strong Bisimilarity and Bisimilarity)**.** Let T be a monad, and $G = (V, B, I)$ and $G' = (V', B', I')$ be T-graphs in $T\text{-}\mathit{DB}^X_Y$. Then G and G' are *strong bisimilar* if there is a bisimulation relation R w.r.t. the endofunctor $T(\mathcal{L}_\epsilon{\times}(\text{-}){+}Y)$ between B and B' such that for any $\&x \in X$, $I(\&x) \mathrel{R} I'(\&x)$.

Now let iter be a uniform iteration operator in $\mathbf{Set}_T$. Then G and G' are *bisimilar* (written as $G \sim G'$) if $\varepsilon\text{-elim}(G)$ and $\varepsilon\text{-elim}(G')$ are strong bisimilar. ☐

Note that the notions of ε-elimination and bisimilarity depend on a given iteration operator, but in this paper we do not refer the dependency in the terminology.

We assume that all monads T in the paper preserve weak-pullbacks, which is just a mild assumption often used in coalgebra theory [Rutten 2000; Sokolova 2005]. Especially, then T preserves injections and finite intersections. Using this assumption, it is easily checked that the strong bisimilarity relation is an equivalence relation on $T\text{-}\mathit{DB}^X_Y$.

It is immediately shown that strong bisimilarity implies bisimilarity from the uniformity of an iteration operator. We use this property in some proofs in the papers. (In fact we can weaken the assumption of uniformity to that of *uniformity with respect to morphisms in* $\mathbf{Set}$ [Simpson and Plotkin 2000, Definition 2.7].)

Thus we gave the definition of bisimilarity, but it is not necessary that every monad has an iteration operator in the Kleisli category. However, as the case of List with CList, there is the case that a monad T is presentable in programming languages, does not have iteration operator, and has an extension T' which might not be presentable in languages but has a uniform iteration operator. We say that T *has an extension T' for ε-elimination* if there is a monad T', an injective monad morphism $\iota : T \hookrightarrow T'$, and a uniform iteration operator in the Kleisli category of T'. Here *monad morphism* is a natural transformation which is compatible with return's, and with lift's (see [Benton et al. 2000], for the detail).

By this, we generalize the definition of bisimilarity:

Definition 11 (Bisimilarity Generalized on Size)**.** Let T be a monad which has an extension T' for ε-elimination. Then there is the embedding $\iota\text{-}\mathit{DB}^X_Y \colon T\text{-}\mathit{DB}^X_Y \hookrightarrow T'\text{-}\mathit{DB}^X_Y$ which maps (V, B, I) to $(V, \iota_{(\mathcal{L}_\epsilon \times V + Y)} \circ B, I)$. For G and G' in $T\text{-}\mathit{DB}^X_Y$, *G and G' are bisimilar* if $\iota\text{-}\mathit{DB}^X_Y(G)$ and $\iota\text{-}\mathit{DB}^X_Y(G')$ are bisimilar in the sense of Definition 10.

By the assumption that ι has injective components and T preserves weak pullback, $\iota\text{-}\mathit{DB}^X_Y$ reflects strong bisimilarity [Sokolova 2005, Theorem 4.3.6], and hence for T-graphs having no ε-edges, strong bisimilarity and the above bisimilarity are equivalent.

Example 12. It is easy to see that the original bisimilarity for unordered graphs is equivalent to the general definition of bisimilarity with $T = P_{\mathrm{fin}}$ and $T' = P_{\mathrm{cnt}}$: countable powerset. For

$f\colon S \to P_{\mathrm{cnt}}(S{+}S')$,

$$iter(f)(s) \overset{\text{def}}{=} \bigcup_{n\in\mathbb{N}} (f_2 \mathbin{\underline{\circ}} (f_1)^n)(s)),$$

where

$$f_1 \overset{\text{def}}{=} [\underline{id}_S, const_{\{\}}] \mathbin{\underline{\circ}} f\colon S \to P_{\mathrm{cnt}}(S)$$

$$f_2 \overset{\text{def}}{=} [const_{\{\}}, \underline{id}_{S'}] \mathbin{\underline{\circ}} f\colon S \to P_{\mathrm{cnt}}(S')$$

and the n-times composition $(f_1)^n$ is that in $\mathbf{Set}_{P_{\mathrm{cnt}}}$.

Similarly, finite multiset monad M_{fin} has an extension for ε-elimination, i.e., *countable multiset monad* M_{cnt}:

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

The iteration operator for M_{cnt} is given with the same formula as that for P_{cnt}.

Also finite probability distribution monad D_{fin} has an extension for ε-elimination, i.e., *countable subprobability distribution monad* $SubD_{\mathrm{cnt}}$: $SubD_{\mathrm{cnt}}(S) \overset{\text{def}}{=}$

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

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 that of the abort 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 [Jacobs 2010b] for the detail.

For $List$ with $CList$, conversely rather we can give the iteration operator in $\mathbf{Set}_{CList}$ with the ε-elimination in Definition 7 in the way described after Definition 9. □

3.4 Bisimilarity for Higher Order Functions

So far we have given the semantics for base types, i.e., bisimilarity for graph types $\mathbf{DB}^X_Y$, and give just equality relations for the other base types. Here we extend such equivalence relations for base types to higher order function types.

It is well known that if we want to lift equivalence relation to function types then we need to switch from the notion of equivalence relation to that of *partial equivalence relation*, i.e., an equivalence relation on some subset of an original set. This is because, now not all functions on $DB\mathrm{f}^X_Y$ are bisimulation generic, so we have to cut out the *subset* consisting of bisimulation generic functions.

Now let us see the formal definition. Let σ be a type of λ^T_{FG}. We define binary logical relation $\sim_\sigma$ from the above equivalence relations on the base types. Let us recall the logical relation $\sim_\sigma$ only on the essential case, i.e., function type: $\sigma = \sigma_1 \to \sigma_2$. By induction hypothesis, we already defined a binary relation $\sim_{\sigma_i}$ on $[\![\sigma_i]\!]$. Then we define a binary relation $\sim_\sigma$ on $[\![\sigma]\!] \overset{\text{def}}{=} [\![\sigma_1]\!] \to [\![\sigma_2]\!]$ as

$$\begin{aligned} f \sim_\sigma f' &\overset{\text{def}}{\iff} \forall x, x' \in [\![\sigma_1]\!].\ (x \sim_{\sigma_1} x' \Rightarrow f(x) \sim_{\sigma_2} f'(x')) \\ &\iff \forall x \in |{\sim_{\sigma_1}}|.\ f(x) \sim_{\sigma_2} f'(x). \end{aligned}$$

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

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

We call a function $f\colon [\![\sigma_1]\!] \to [\![\sigma_2]\!]$ *(higher order) bisimulation generic* if f is in $|{\sim_{\sigma_1\to\sigma_2}}|$, i.e.,

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

Then by the Basic Lemma of logical relation, interpretations of all the terms are bisimulation generic if interpretations of all the constants are bisimulation generic; and then we obtain a model of λ^T_{FG} in the cartesian closed category $\mathbf{Set}$, see the textbook [Mitchell 1996] for the detail. Note that the above lifting to function types is possible for any equivalence relations such as strong bisimilarity.

In the next section we show the bisimulation genericity for all constants.

4. UnCAL Generalized with Monad

In this section we give interpretations of terms of λ^T_{FG} and show their bisimulation genericity. As explained in the last of the previous section, it is enough to consider only constants; we will see graph constructors, structural recursion, and sibling transformations. In the last place, we see how to define syntax of λ^T_{FG}.

4.1 Graph Constructors

UnCAL and $\lambda^{List}_{\mathrm{FG}}$ respectively have nine graph constructors, as in Section 2.2.2, by which we can represent all finite graphs. Here we define such graph constructors for T-graphs by which we can represent all finite T-graphs.

Among the nine graph constructors of $\lambda^{List}_{\mathrm{FG}}$, [] and $+\!\!+$ are inherent in the list monad $List$, and the other seven constructors are common for all monads. $List(S)$ is free monoid (generated by a set S), which has nullary and binary operations: unit and multiplication. The nullary graph constructor [] and binary graph constructor $+\!\!+$ correspond to the unit and the multiplication, respectively.

First we define the seven common graph constructors for T-graphs.

Definition 13 (Common Graph Constructors)**.**

- For $G = (V, B, I) \in DB_Y$,

$$\begin{aligned} \langle l : G\rangle &\overset{\text{def}}{=} (V \cup \{v_0 : \text{fresh}\},\ B',\ \{\&\mapsto v_0\}) \in DB_Y \\ B' &\overset{\text{def}}{=} B \cup \{v_0 \mapsto return\big(\mathsf{Edge}\,(l, I(\&))\big)\}. \end{aligned}$$

- For $G = (V, B, I) \in DB_Y$,

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

- For $\&y \in Y$,

$$\langle \&y\rangle \overset{\text{def}}{=} (\{\&\},\ \{\& \mapsto return(\mathsf{Outm}\,(\&y))\},\ id_{\{\&\}}) \in DB_Y.$$

- $() \overset{\text{def}}{=} (\emptyset,\ \emptyset,\ id_\emptyset) \in DB^\emptyset_Y$.
- For $G = (V, B, I) \in DB^X_Y$ and $G' = (V', B', I') \in DB^{X'}_Y$ such that $X \cap X' = \emptyset$,

$$\begin{aligned} G \oplus G' &\overset{\text{def}}{=} (V{+}V',\ B'',\ I{+}I') \in DB^{X\cup X'}_Y \\ B'' &\overset{\text{def}}{=} [T(\mathcal{L}_\epsilon{\times}(in_l){+}Y) \circ B,\ T(\mathcal{L}_\epsilon{\times}(in_r){+}Y) \circ B'] \\ &\quad : V{+}V' \to T(\mathcal{L}_\epsilon{\times}(V{+}V'){+}Y). \end{aligned}$$

- For $G = (V, B, I) \in DB^X_Y$ and $G' = (V', B', I') \in DB^Y_Z$, $G \mathbin{@} G' \overset{\text{def}}{=} (V{+}V',\ B'',\ in_l \circ I) \in DB^X_Z$ where

$$B''(in_l(v)) \overset{\text{def}}{=} lift(f)(B(v))$$
$$\left(\begin{array}{l} f\colon \mathcal{L}_\epsilon{\times}V{+}Y \to T(\mathcal{L}_\epsilon{\times}(V{+}V'){+}Z) \\ \quad \mathsf{Edge}\,(l, v) \mapsto return(\mathsf{Edge}\,(l, in_l(v))) \\ \quad \mathsf{Outm}\,(\&y) \mapsto (T(\mathcal{L}_\epsilon{\times}(in_r){+}Z))(B'(I'(\&y))) \end{array}\right)$$
$$B''(in_r(v')) \overset{\text{def}}{=} (T(\mathcal{L}_\epsilon{\times}(in_r){+}Z))(B'(v')).$$

- For a graph $G = (V, B, I) \in DB^X_{X\cup Y}$ where $X \cap Y = \emptyset$, $\mathbf{cycle}(G) \overset{\text{def}}{=} (V, B', I) \in DB^X_Y$ where

$$B'(v) \overset{\text{def}}{=} T(f)(B(v)).$$
$$\begin{pmatrix} f : \mathcal{L}_\epsilon \times V + (X \cup Y) \to \mathcal{L}_\epsilon \times V + Y \\ \mathsf{Edge}\,(l, v) \mapsto \mathsf{Edge}\,(l, v) \\ \mathsf{Outm}\,(\&x) \mapsto \mathsf{Edge}\,(\varepsilon, I(\&x)) \\ \mathsf{Outm}\,(\&y) \mapsto \mathsf{Outm}\,(\&y) \end{pmatrix}$$

□

In the definition of $B''(in_l(v))$ in the definition of @, by replacing $lift(f)(B(v))$ with $T(f')(B(v))$ where

$$\begin{aligned} f' \colon \mathcal{L}_\epsilon \times V + Y &\to \mathcal{L}_\epsilon \times (V + V') + Z \\ \mathsf{Edge}\,(l, v) &\mapsto \mathsf{Edge}\,(l, in_l(v)) \\ \mathsf{Outm}\,(\&y) &\mapsto \mathsf{Edge}\,\big(\varepsilon, I'(\&y)\big) \end{aligned}$$

we obtain @$'$ operator, which is bisimilar to @ operator. With @ operator, we find that we do not need ε-edge for graph constructors except for **cycle**, but @$'$ is useful to postpone extra calculation of @, which is in effect equivalent to one step of ε-elimination.

We note that the multi-rootedness is semantically the same as power of sets of graphs (or tuple of graphs): $DB_{\mathrm{f}}{}_Y^X \cong (DB_{\mathrm{f}\,Y})^X$. The function $DB_{\mathrm{f}}{}_Y^X \to (X \to DB_{\mathrm{f}\,Y})$ is $G \mapsto (\&x \mapsto \langle \&x \rangle \; @ \; G)$, and the inverse is

$$f \mapsto (\&x_1 := f(\&x_1)) \oplus ... \oplus (\&x_n := f(\&x_n))$$

when $X = \{\&x_1, ..., \&x_n\}$. Still multi-rootedness is useful for efficient implementation, i.e., to share bisimilar nodes and to save the size of graphs as small as possible.

Now we define the other graph constructors which depend on each monad T. For Proposition 19 and so on, we finally assume that a monad T is finitary; though we can postpone the definition of finitary monad till Proposition 19, we first explain finitary monad so that the reader has more concrete intuition in the following definitions and propositions.

A *finitary monad* on $\mathbf{Set}$ is a monad T on $\mathbf{Set}$ such that the functor T preserves all directed colimits in $\mathbf{Set}$. We use this property only in the following form (and an equivalent condition explained soon): 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 $[4,1,6,4,6,4] \in List(\mathbb{N})$, we have a finite subset $\{4,1,6\} \subseteq \mathbb{N}$ and $[4,1,6,4,6,4] \in List(\{4,1,6\})$.

Next we recall one equivalent condition of finitary monad T: shortly speaking, T is generated by finite arity algebraic operations.

For a monad T on $\mathbf{Set}$, an element $f \in T(A)$ can be seen as an $|A|$-ary algebraic operation on $T(S)$ for any $S \in \mathbf{Set}$: for $g \in T(S)^A$,

$$\widehat{f}(g) \overset{\text{def}}{=} lift(g)(f) \in T(S).$$

For example, when $T = P_{\mathrm{fin}}$, for $f = \{0,1\} \in P_{\mathrm{fin}}(\{0,1\})$ and $g_0, g_1 \in P_{\mathrm{fin}}(S)$,

$$\widehat{f}(g_0, g_1) = lift(0 \mapsto g_0, 1 \mapsto g_1)(\{0,1\}) = g_0 \cup g_1,$$

i.e., $\widehat{\{0,1\}} = \cup$.

For a monad T, let us take a family of sets $\Sigma(n) \subseteq T(n)$ ($n \in \mathbb{N}$), where we regard n as the set $\{0, ..., n-1\}$. Let us define $T_\Sigma^{(i)}(S) \subseteq T(S)$ ($S \in \mathbf{Set}$) by induction on $i \in \mathbb{N}$ as below:

$$T_\Sigma^{(0)}(S) \overset{\text{def}}{=} return(S) \subseteq T(S)$$
$$T_\Sigma^{(i+1)}(S) \overset{\text{def}}{=} \{\widehat{f}(g) \in T(S) \mid m \in \mathbb{N},\ g \in (T_\Sigma^{(i)}(S))^m\ f \in \Sigma(m)\}.$$

Then a monad T is finitary if and only if there is a Σ such that

$$T(S) = \bigcup_{i \in \mathbb{N}} T_\Sigma^{(i)}(S),$$

and in this case, we call such Σ a *family of signature sets of* T and we call elements in $\Sigma(n)$ *signatures*.

Example 14. The three monads P_{fin}, M_{fin}, and $List$ correspond to (upper) semilattice, commutative monoid, and monoid, respectively; then their zero ary operations and binary operations correspond to signatures as below.

For P_{fin},

$$\begin{aligned} \Sigma(0) &\overset{\text{def}}{=} \{\{\}\} && \subseteq P_{\mathrm{fin}}(0) \\ \Sigma(2) &\overset{\text{def}}{=} \{\{0,1\}\} && \subseteq P_{\mathrm{fin}}(2) \\ \Sigma(n) &\overset{\text{def}}{=} \emptyset && \text{(for other } n\text{)}. \end{aligned}$$

For $\{\} \in \Sigma(0)$, $\widehat{\{\}} = \{\} \in P_{\mathrm{fin}}(S)$, and as seen above,

$$\widehat{\{0,1\}} = \cup \colon P_{\mathrm{fin}}(S)^2 \to P_{\mathrm{fin}}(S).$$

Then any element in $P_{\mathrm{fin}}(S)$ can be represented as a composition of these operators and singletons, i.e., elements in $return(S)$: e.g. $\{a,b,c\} = (\{a\} \cup \{b\}) \cup \{c\}$.

For M_{fin}, similarly,

$$\begin{aligned} \Sigma(0) &\overset{\text{def}}{=} \{\{\}\} && \subseteq M_{\mathrm{fin}}(0) \\ \Sigma(2) &\overset{\text{def}}{=} \{\{0,1\}\} && \subseteq M_{\mathrm{fin}}(2) \\ \Sigma(n) &\overset{\text{def}}{=} \emptyset && \text{(for other } n\text{)}. \end{aligned}$$

Here $\{\}$ and $\{0,1\}$ are regarded as multisets.

For $List$, similarly again,

$$\begin{aligned} \Sigma(0) &\overset{\text{def}}{=} \{[]\} && \subseteq List(0) \\ \Sigma(2) &\overset{\text{def}}{=} \{[0,1]\} && \subseteq List(2) \\ \Sigma(n) &\overset{\text{def}}{=} \emptyset && \text{(for other } n\text{)}. \end{aligned}$$

For D_{fin},

$$\begin{aligned} \Sigma(2) &\overset{\text{def}}{=} \{sig_r \colon 2 \to [0,1] \mid r \in [0,1]\} && \subseteq D_{\mathrm{fin}}(2) \\ \Sigma(n) &\overset{\text{def}}{=} \emptyset && \text{(for other } n\text{)} \end{aligned}$$

where sig_r is defined as $sig_r(0) \overset{\text{def}}{=} r$ and $sig_r(1) \overset{\text{def}}{=} 1-r$. Then for $\phi_1, \phi_2 \colon S \to [0,1]$ in $D_{\mathrm{fin}}(S)$, $\widehat{sig_r}(\phi_0, \phi_1) \in D_{\mathrm{fin}}(S)$ is $\widehat{sig_r}(\phi_0, \phi_1) \colon S \to [0,1]$ such that

$$\widehat{sig_r}(\phi_0, \phi_1)(s) = r \cdot \phi_0(s) + (1-r) \cdot \phi_1(s).$$

Now recall that "singletons" in $D_{\mathrm{fin}}(S)$, i.e., elements in $return(S)$ are given as the Dirac delta functions δ_s as in Example 3. Then it is easy to see that the above family becomes a family of signature sets; for example, $\phi \colon \mathbb{N} \to [0,1]$ in $D_{\mathrm{fin}}(\mathbb{N})$ such that

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

can be represented by signatures and "singletons" as

$$\begin{aligned} \phi &= \frac{1}{2}\delta_0 + \frac{1}{6}\delta_1 + \frac{1}{3}\delta_2 \\ &= \frac{1}{2}\delta_0 + \frac{1}{2}(\frac{1}{3}\delta_1 + \frac{2}{3}\delta_2) \\ &= \widehat{sig_{\frac{1}{2}}}(\delta_0, \widehat{sig_{\frac{1}{3}}}(\delta_1, \delta_2)). \end{aligned}$$

□

The above family of signature sets of D_{fin} has infinitely many signatures. Still, when implementing λ_{FG}^T in a programming language, one can represent it as the image of a function. For example, first let us interpret $D_{\mathrm{fin}}(S)$ as $\texttt{FinSet}(S \times \texttt{Float})$ (or with finite multiset instead of finite set). Here a distribution $f \in D_{\mathrm{fin}}(S)$

is regarded as a relation rather than a function, then a probability $f(s) \in [0,1]$ can be regarded as the sum of the multi values of $f(s)$. Then the family of signature sets Σ for D_{fin} is presented as $\texttt{sig}\colon \texttt{Float} \to D_{\text{fin}}(\texttt{Nat})$ defined as below:

```
  let sig r = {(0,r), (1,1-r)}
```

Now we define a graph constructor for T-graphs for each algebraic operator of T, i.e., for each element in $T(A)$. Before this, note that in the definitions of graph and common graph constructors, we can easily extend set of marker from finite to arbitrary set A, and $\oplus$ from binary operator to A-ary operator: $\oplus\colon (DB^X_Y)^A \overset{\cong}{\to} DB^{A\times X}_Y$. These are used in the following definition, but the reader who is interested only in finite A's—which are enough for implementable language—may apply the following simply to the case $A = n = \{0, ..., n{-}1\}$.

Definition 15 (T-Algebraic Graph Constructors)**.** Let T be a monad on $\mathbf{Set}$. For $f \in T(A)$ ($A \in \mathbf{Set}$), and a finite set X, let $G^X_f \overset{\text{def}}{=} (X, B, id) \in DB^X_{A\times X}$ where

$$B(\&x) \overset{\text{def}}{=} T(g)(f) \in T(\mathcal{L}_\epsilon{\times}X{+}A{\times}X),$$
$$g\colon A \to \mathcal{L}_\epsilon{\times}X{+}(A{\times}X)$$
$$\&a \mapsto \mathsf{Outm}\,((\&a, \&x))\,.$$

Then we define a T-*algebraic graph constructor* $\overline{f}\colon (DB^X_Y)^A \to DB^X_Y$. For $(G_i)_{i\in A} \in (DB^X_Y)^A$, $\overline{f}((G_i)_i) \overset{\text{def}}{=} G^X_f \;@\; (\oplus((G_i)_i))$.

We call T-algebraic graph constructors and the common graph constructors *graph constructors*.

Example 16. When $T = \mathit{List}$, let us consider $f = [0,1] \in \Sigma(2)$ in Example 14. Let us see the picture of $G_1 +\!\!+\, G_2$ in Figure 3. This is the @$'$ version of $\overline{[0,1]}(G_1, G_2)$. The top m-roots are $G_{[0,1]}$ and the ε-edges are those by @$'$; and one can find that the juxtaposition of two graphs G_1 and G_2 in $G_1 +\!\!+\, G_2$ are the same as that in $G_1 \oplus G_2$ in the same figure.

Proposition 17. *Let T be a monad on $\mathbf{Set}$ with an extension for ε-elimination. For $f \in T(A)$ ($A \in \mathbf{Set}$), and $f_i \in T(A_i)$ ($i \in A, A_i \in \mathbf{Set}$), let*

$$\underline{\textstyle\coprod}_{i\in A} f_i\colon A(\cong \textstyle\coprod_{i\in A} 1) \to T(\textstyle\coprod_{i\in A} A_i)$$

be the coproduct of the morphisms $f_i\colon 1 \to T(A_i)$ in the Kleisli category $\mathbf{Set}_T$. Then

$$\overline{f}((\overline{f_i}(\text{-}))_{i\in A})\colon \textstyle\prod_{i\in A}(DB^X_Y)^{A_i} \to DB^X_Y$$

is bisimilar to

$$\overline{\mathit{lift}(\underline{\textstyle\coprod}_{i\in A} f_i)(f)}\colon (DB^X_Y)^{\coprod_{i\in A} A_i} \to DB^X_Y,$$

via the isomorphism

$$\textstyle\prod_{i\in A}(DB^X_Y)^{A_i} \cong (DB^X_Y)^{\coprod_{i\in A} A_i}.$$

Hence, if T is further a finitary monad with a family of signature sets Σ, the graph constructor $\overline{f}$ of any $f \in T(A)$ is a composition of the graph constructors of a finite number of some signatures in Σ.

At the same time, when we add graph constructors of all the signatures in one family of signature sets to a calculus, then expressive power on graph constructors are the same independently from the choice of families of signature sets.

Proof. For $G^i_{i'} = (V^i_{i'}, B^i_{i'}, I^i_{i'})$ ($i \in A, i' \in A_i$),

$$\overline{f}((\overline{f_i}((G^i_{i'})_{i'}))_i).\mathrm{V} = X{+}\Sigma_{i\in A}(X{+}\Sigma_{i'\in A_i} V^i_{i'})$$

and

$$\overline{\mathit{lift}(\underline{\textstyle\coprod}_{i\in A} f_i)(f)}((G^i_{i'})_{(i,i')}).\mathrm{V} = X{+}\Sigma_{(i,i')\in\coprod_{i\in A} A_i} V^i_{i'}.$$

In the former there are extra A-copy of X, i.e., $\Sigma_{i\in A}(X)$ than the latter; but with @ (rather than @$'$) in Definition 15, the nodes in $\Sigma_{i\in A}(X)$ in the former are not reachable from the roots in the former graph, so can be ignored.

The second and third points are immediate from the first point by the definition of a family of signature sets of a finitary monad. □

This proposition also implies that T-algebraic graph constructors obey the same axioms as those of algebra of T. For example, finite powersets are free algebras of upper semilattice, hence the graph constructors $\{\}$ and (-) $\cup$ (-) satisfy all axioms of upper semilattice: i.e., associativity, unitality, commutativity, and idempotency.

Proposition 18 (Bisimulation Genericity of Graph Constructors)**.** *All the graph constructors (including @$'$) are strong-bisimulation generic and also are bisimulation generic.*

Proof. It is obvious that they are strong-bisimulation generic. Then, for a graph constructor f, prove that $\varepsilon\text{-elim}(f(G_1, ..., G_n))$ is bisimilar to $\varepsilon\text{-elim}(f(\varepsilon\text{-elim}(G_1), ..., \varepsilon\text{-elim}(G_n)))$. (For the case of $\mathbf{cycle}$, use unfolding axiom of iteration operator.) This implies that strong-bisimulation genericity implies bisimulation genericity. □

The next proposition is the most important property of the graph constructors, for which we require T to be finitary.

Proposition 19 (Full-Representability)**.** *Let T be a finitary monad on $\mathbf{Set}$ which has an extension for ε-elimination. Any finite T-graphs can be constructed by the graph constructors.*

Proof. The proof is basically the same as that for UnCAL [Buneman et al. 2000]. See Appendix D for the detail. □

4.1.1 Remark on Representability of Monads

We give an important remark on "representability" of monads in languages, which explains why we need care to infinite width for List-graphs, and why we do not need such special care for P_{fin}-graph.

For T with an extension for ε-elimination T', we defined ε-elimination for any T-graphs, but in fact it is enough to define ε-elimination for finite T-graphs, since our languages target only finite graphs. If T is finitary, in order to define ε-elimination only for finite T-graphs, without loss of generality we can replace the extension T' with its *finitary part* $T'|_{\text{fin}}$:

$$T'|_{\text{fin}}(S) \overset{\text{def}}{=} \textstyle\bigcup_{S'\in P_{\text{fin}}(S)} T'(S').$$

(See Appendix C for the detail why generality is not lost.)

Then, $(P_{\text{cnt}})|_{\text{fin}}$ is equal to P_{fin}, hence we do not need P_{cnt} for $\lambda^{P_{\text{fin}}}_{\text{FG}}$ anymore. On the other hand, $\mathit{CList}|_{\text{fin}}(S)$ consists of countable lists l such that the number of elements in S which occur in one l are finite. For example, $[0,1,0,1,...] \in \mathit{CList}|_{\text{fin}}(\mathbb{N})$, but $[0,1,2,3,...] \notin \mathit{CList}|_{\text{fin}}(\mathbb{N})$, Thus $\mathit{CList}|_{\text{fin}}$ is far different from List; we can not avoid the use of the notion of countable linear ordered set. (However the authors do not know if for arbitrary countable linear ordered set L there is an *finite List*-graph G whose ε-elimination involves L; though we found such G as *infinite List*-graphs as in Appendix A. As in Figure 6, it is certain that a dense countable linear ordered set (and more countable linear ordered sets concatenated further) occur.)

For the other two examples,

$$(M_{\text{cnt}})|_{\text{fin}}(S) = \{\phi\colon S{\to}\mathbb{N}{\cup}\{\infty\} \mid \phi^{-1}(\mathbb{N}{-}\{0\}) \text{ is finite}\},$$

and $(SubD_{\mathrm{cnt}})|_{\mathrm{fin}}$ is the *finite subprobability distribution monad* $SubD_{\mathrm{fin}}$:

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

Here $(M_{\mathrm{cnt}})|_{\mathrm{fin}}$ and $(SubD_{\mathrm{cnt}})|_{\mathrm{fin}}$ are representable in a language for implementing $\lambda^{T}_{\mathrm{FG}}$ in a similar way to that after Example 14, while it seems difficult to represent whole of $CList|_{\mathrm{fin}}$, because of the notion of countable linear ordered set.

Note the difference between being finitary and being "representable", and also note the difference between being "representable" of T and that of T'. (On the other hand, recall that as above if T is finitary then T' can be finitary.) As above, it is not clear that being finitary of a monad implies that the monad is "representable" in a language; and being finitary of T is used to prove the property that every term in $\lambda^{T}_{\mathrm{FG}}$ preserves finiteness (of nodes) of graphs, while "representability" of T is needed for syntax of graph constructors and implementation of the graph model $B\colon V \to T(\mathcal{L}_\epsilon{\times}V{+}Y)$. On the other hand, "representability" of T' is needed for representation of graphs without ε-edges, which is important if, in an application, graphs observable from users of $\lambda^{T}_{\mathrm{FG}}$ should be ε-edge free; in order to resolve this problem, we gave the effective procedure in Section 3.2 for the case of $List$ with $CList$.

4.2 Structural Recursion

Now we give a general definition of the structural recursion, which is the most important transformation method of $\lambda^{T}_{\mathrm{FG}}$. In [Buneman et al. 2000], there are two semantics for the structural recursion: bulk semantics and memoized recursive semantics. Here we generalize the bulk semantics.

A picture of the bulk semantics for ordered graphs is given in Figure 7. As seen in the picture, with bulk semantics first we calculate the application of a given input function f to a pair of each edge and its following subgraph of an input graph. Then we connect the results in keeping with the shape of the original graph, using ε-edges.

Recall the record notation $G = (G.\mathrm{V}, G.\mathrm{B}, G.\mathrm{I})$, which is used in the following.

Definition 20 (Bulk Semantics of Structural Recursion)**.** Let T be a monad on $\mathbf{Set}$. For a function $e\colon \mathcal{L}{\times}DB_Y \to DB^{Z}_{Z}$, a *structural recursion function* $\mathbf{srec}(e)\colon DB^{X}_{Y} \to DB^{Z\times X}_{Z\times Y}$ is defined as the following.

For $G = (V, B, I) \in DB^{X}_{Y}$, $\mathbf{srec}(e)(G) \stackrel{\mathrm{def}}{=} (V', B', I') \in DB^{Z\times X}_{Z\times Y}$ where

- $V' \stackrel{\mathrm{def}}{=} (Z{\times}V) + (\Sigma_{(l,v')\in\mathcal{L}\times V}\, e(l, G|_{v'}).\mathrm{V})$
  (Here, an element $(\&z, v)$ in the left set $Z{\times}V$—Z-copy of nodes of G—is a "hub" of the new graph, for this case we use the label $\mathsf{Hub}(\&z, v)$ below. While, on the right of "$+$" and further in the case of (l, v') of the direct sum "Σ", a node u in $e(l, G|_{v'}).\mathrm{V}$ is "a piece of bulk"; each l-labeled edge to v' in G is replaced with this subgraph $e(l, G|_{v'}).\mathrm{V}$, which is represented by a dotted box in Figure 7. For this case, we use the label $\mathsf{Bulk}_{(l,v')}(u)$.)
- $B'\colon (Z{\times}V) + (\Sigma_{(l,v')}\, e(l, G|_{v'}).\mathrm{V}) \to$
  $T\Big(\mathcal{L}_\epsilon{\times}\big((Z{\times}V) + (\Sigma_{(l,v')}\, e(l, G|_{v'}).\mathrm{V})\big) + (Z{\times}Y)\Big)$
  $\mathsf{Hub}(\&z, v) \;\mapsto T(g)\big(\mathit{lift}(\lambda x.(\&z, x))(B(v))\big)$

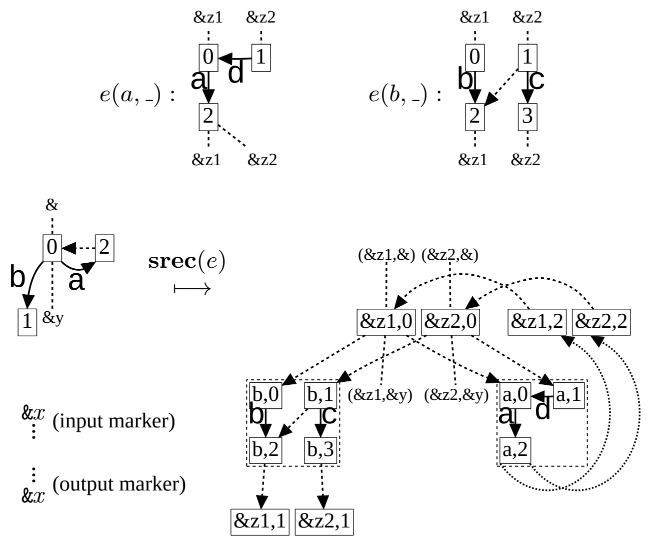


Figure 7. Bulk Semantics of Structural Recursion

$$\left(\begin{array}{l}
g\colon Z{\times}(\mathcal{L}_\epsilon{\times}V{+}Y) \to \\
\qquad\qquad \mathcal{L}_\epsilon{\times}\big((Z{\times}V) + (\Sigma_{(l,v')}\, e(l, G|_{v'}).\mathrm{V})\big) + (Z{\times}Y) \\
(\&z, \mathsf{Edge}\,(l, v')) \;\;\mapsto \\
\qquad (\text{if } l = \varepsilon\,) \quad \mathsf{Edge}\,\big(\varepsilon, \mathsf{Hub}(\&z, v')\big) \\
\qquad (\text{if } l \neq \varepsilon\,) \quad \mathsf{Edge}\,\big(\varepsilon, \mathsf{Bulk}_{(l,v')}((e(l, G|_{v'}).\mathrm{I})(\&z))\big) \\
(\&z, \mathsf{Outm}\,(\&y)) \quad \mapsto \mathsf{Outm}\,((\&z, \&y))
\end{array}\right)$$

$$\mathsf{Bulk}_{(l,v')}(u) \;\;\mapsto\;\; T(h)\big((e(l, G|_{v'}).\mathrm{B})(u)\big)$$

$$\left(\begin{array}{l}
h\colon \mathcal{L}_\epsilon{\times}(e(l, G|_{v'}).\mathrm{V}){+}Z \to \\
\qquad\qquad \mathcal{L}_\epsilon{\times}\big((Z{\times}V) + (\Sigma_{(l,v')}\, e(l, G|_{v'}).\mathrm{V})\big) + (Z{\times}Y) \\
\mathsf{Edge}\,(l', u') \;\;\mapsto\; \mathsf{Edge}\,\big(l', \mathsf{Bulk}_{(l,v')}(u')\big) \\
\mathsf{Outm}\,(\&z) \quad\;\; \mapsto\; \mathsf{Edge}\,\big(\varepsilon, \mathsf{Hub}(\&z, v')\big)
\end{array}\right)$$

- $I'\colon\; Z{\times}X \;\to\; (Z{\times}V) + (\Sigma_{(l,v')}e(l, G|_{v'}).\mathrm{V})$
  $(\&z, \&x) \;\;\mapsto\;\; \mathsf{Hub}(\&z, I(\&x))$ $\square$

Next we see how structural recursion preserves finiteness of graphs.

Proposition 21. *Let T be a finitary monad. Structural recursion function maps finite graphs to finite graphs; more precisely, for $e\colon \mathcal{L}{\times}DB_{\mathrm{f}\,Y} \to DB_{\mathrm{f}}\,^{Z}_{Z}$ and a finite T-graph G, the accessible part of $\mathbf{srec}(e)(G)$ is finite.*

Proof. Since T is finitary, for each element $v \in V$, there is a finite subset $E_v \subseteq \mathcal{L}_\epsilon{\times}V$ such that $B(v) \in T(E_v{+}Y)$. Let

$$E \stackrel{\mathrm{def}}{=} \big(\textstyle\bigcup_{v\in V} E_v\big) \cap (\mathcal{L}{\times}V) \subseteq \mathcal{L}{\times}V,$$

then since V is finite, so is E, and B is decomposed through $T(E{+}Y) \subseteq T(\mathcal{L}_\epsilon{\times}V{+}Y)$. Then the accessible part of $\mathbf{srec}(e)(G)$ is included in the finite set

$$V'' \stackrel{\mathrm{def}}{=} (Z{\times}V) + (\Sigma_{(l,v')\in E}\, e(l, G|_{v'}).\mathrm{V}) \subseteq V',$$

i.e., $I'(Z{\times}X) \subseteq V''$ and $B'(V'') \subseteq V''$. $\square$

Now recall the notion of bisimilarity for higher order functions. The following is stronger result than that proved in [Buneman et al. 2000] even when $T = P_{\mathrm{fin}}$, because here bisimulation genericity is proved also on the first argument e, while in [Buneman et al. 2000] it is proved only on the second argument G. This is the key why we can extend from UnCAL to the higher order calculus $\lambda_{\mathrm{FG}}^{P_{\mathrm{fin}}}$.

Theorem 22. *Let T be a finitary monad having an extension for ε-elimination T'. Structural recursion* $\mathbf{srec}$ *is bisimulation generic, i.e., if*

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

$$G_1 \sim G_2 \ \in\ T\text{-}DB_{\mathrm{f}\,Y}^{\,X},$$

then

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

Proof. Basically the proof is similar to that of Proposition 18 but a bit subtle, because ε-elim embed $T\text{-}DB_{\mathrm{f}}$ into $T'\text{-}DB$, while higher order function types make inclusion relations between types opposite. So, first we show that the structural recursion is defined "uniform" on size of graphs: i.e., let

$$i \overset{\text{def}}{=} (\iota\text{-}DB_Y^X)|_{T\text{-}DB_{\mathrm{f}\,Y}^{\,X}} \colon T\text{-}DB_{\mathrm{f}\,Y}^{\,X} \to T'\text{-}DB_Y^X,$$

then for

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

$$e' \colon \mathcal{L}{\times}T'\text{-}DB_Y \to T'\text{-}DB_Z^Z,$$

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

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

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

This reduces the setting of the theorem to the case when $\iota\colon T \to T'$ is the identity and also the case of infinite graph ($T\text{-}DB$ rather than $T\text{-}DB_{\mathrm{f}}$), because for any

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

there are

$$e_1' \sim e_2' \colon \mathcal{L}{\times}T'\text{-}DB_Y \to T'\text{-}DB_Z^Z$$

such that $e_i' \circ (\mathcal{L}{\times}i) = i \circ e_i$.

Now, in this reduced setting, we can prove commutativity of $\mathbf{srec}$ with ε-elim:

$$\varepsilon\text{-elim}^{Z\times V}(\varepsilon\text{-elim}^{\Sigma_{(l,v')}e(l,G|_{v'}).\mathrm{V}}(\mathbf{srec}(e)(G))) \\ \sim_{\mathrm{s}} \quad \mathbf{srec}(\varepsilon\text{-elim} \circ e)(\varepsilon\text{-elim}(G)).$$

Here, while ε-elim eliminates ε-edges in $v_0 \xrightarrow{\varepsilon} v_1 ... \xrightarrow{\varepsilon} v_n \xrightarrow{a} u$ and adds $v_0 \xrightarrow{a} u$, $\varepsilon\text{-elim}^W$ does so only if v_0, ..., v_n are in W.

Then it is sufficient in order to conclude the proof to prove that if

$$e_1 \sim_{\mathrm{s}} e_2 \colon \mathcal{L}{\times}T\text{-}DB_Y \to T\text{-}DB_Z^Z, \text{ and}$$

$$G_1 \sim_{\mathrm{s}} G_2 \ \in\ T\text{-}DB_Y^X,$$

then

$$\mathbf{srec}(e_1)(G_1) \sim_{\mathrm{s}} \mathbf{srec}(e_2)(G_2) \ \in\ T\text{-}DB_{Z\times Y}^{Z\times X}.$$

We can define a needed bisimulation relation just as we defined the set of nodes of $\mathbf{srec}(e)(G)$ in the bulk semantics. ☐

4.3 Sibling Transformation

So far we generalized all of UnCAL, parameterizing with monads. However, when we take a monad other than powerset monad, say, the list monad for ordered graph, we need more expressive power than what we got so far: i.e., graph transformations on sibling dimension.

Here we introduce two term constructors for sibling transformation: first *local sibling transformation* $\mathbf{l\text{-}sbl}(f)$ and then *uniform sibling transformation* $\mathbf{u\text{-}sbl}(f)$.

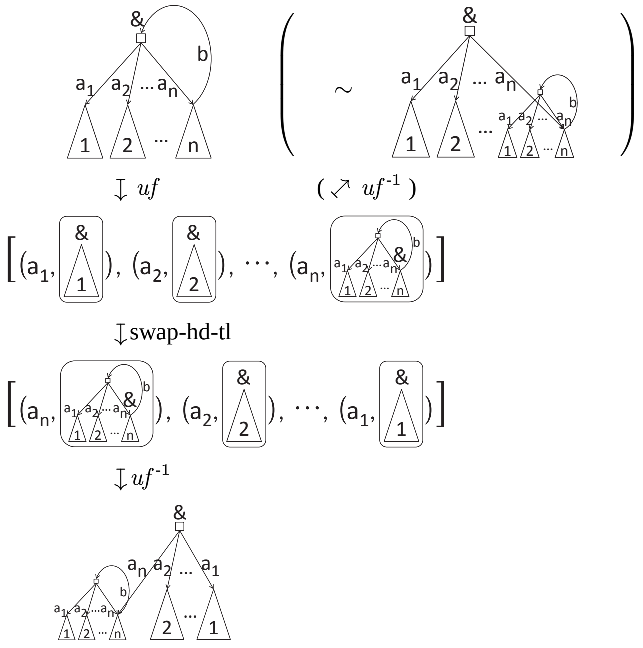


Figure 8. $\mathbf{l\text{-}sbl}$(swap-hd-tl) $(= uf^{-1} \circ$ swap-hd-tl $\circ\, uf)$

The two functions $\mathbf{l\text{-}sbl}(f)$ and $\mathbf{u\text{-}sbl}(f)$ can be applied only to graphs without ε-edge. Hence we take ε-elimination of G before evaluating $\mathbf{l\text{-}sbl}(f)(G)$ or $\mathbf{u\text{-}sbl}(f)(G)$. For T such that its extension T' is not presentable in a programming language (such as $List$ with $CList$), we check if the ε-elimination of a given graph results in a T-graph—by the procedure in Section 3.2 for ordered graphs—and if it is impossible, put an error. If T' is presentable in a language such as multiset, then we consider the T' as an instance of T in the following. Thus below we consider a finitary monad T and T-graphs without ε-edges. (In Example 23, we see why we avoid ε-edges.)

4.3.1 Local Sibling Transformation

Local sibling transformation

$$\mathbf{l\text{-}sbl}(f) \colon DB_{\mathrm{f}\,Y} \to DB_{\mathrm{f}\,Y}$$

for

$$f \colon T(\mathcal{L}{\times}DB_{\mathrm{f}\,Y}{+}Y) \to T(\mathcal{L}{\times}DB_{\mathrm{f}\,Y}{+}Y)$$

manipulates branches of the root node of an input graph in sibling direction. Before apply $\mathbf{l\text{-}sbl}$, we take 1-step unfolding of the input graph, which we now define, so we do not have to mind cycles which the root might belong to.

We define the 1-step unfolding

$$uf \colon DB_{\mathrm{f}\,Y} \to T(\mathcal{L}{\times}DB_{\mathrm{f}\,Y}{+}Y),$$

where note that $\mathcal{L}$ has no ε since we consider graph having no ε-edges. First, for a graph $G = (V, B, I) \in DB_Y^X$, and a node $v \in V$, we define

$$G|_v \overset{\text{def}}{=} (V, B, \{\& \mapsto v\}) \in DB_Y.$$

Now for $G = (V, B, I) \in DB_{\mathrm{f}\,Y}$,

$$uf(G) \overset{\text{def}}{=} T(\mathcal{L}{\times}f{+}Y)(B(I(\&)))$$

where $f \overset{\text{def}}{=} G|_{(\text{-})} \colon V \to DB_{\mathrm{f}\,Y}$. A picture of uf can be seen in Figure 8.

This coalgebra uf is called *final locally finite coalgebra*, and by the results [Adámek et al. 2006, Theorem 3.3] with [Milius 2010, Corollary III.15], we can show that for any finitary monad T, this uf is isomorphic up to bisimilarity and bisimulation generic. For example, when $T = List$,

$$uf^{\text{-}1} = \mathbf{foldr}(+\!\!+, []) \circ List([\langle \text{-} : \text{-} \rangle, \langle \text{-} \rangle]).$$

See Appendix D for the detail of $uf^{\text{-}1}$.

Then **l-sbl** is defined as below: for

$$f \colon T(\mathcal{L} \times DB_{\mathrm{f}\,Y} + Y) \to T(\mathcal{L} \times DB_{\mathrm{f}\,Y} + Y)$$

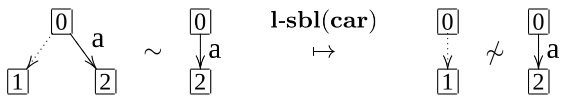
and $G \in DB_{\mathrm{f}\,Y}$,

$$\textbf{l-sbl}(f)(G) \overset{\text{def}}{=} (uf)^{\text{-}1}(f(uf(G))).$$

A picture of an example of local sibling transformation is given in Figure 8, where swap-hd-tl is the list function which swaps the head and tail and maps nil to nil.

Since we take ε-elimination of G before applying $\textbf{l-sbl}(f)$ to G, and since uf is bisimulation generic, it is obvious that **l-sbl** is (higher order) bisimulation generic. On the other hand, if we apply $\textbf{l-sbl}(f)$ to G having ε-edges, bisimulation genericity does not necessarily hold:

Example 23. Let us consider **car** defined as $\mathbf{car}([]) \overset{\text{def}}{=} []$ and $\mathbf{car}(a :: as) = [a]$. Then if we apply $\textbf{l-sbl}(\mathbf{car})$ to the two bisimilar graphs below left, then the results below right are not bisimilar.



□

We remark that, when taking ε-elimination of G before the application $\textbf{l-sbl}(f)(G)$, it is enough to eliminate only ε-edges from the root node of G—by which all branches of the root node become non-ε edges or output markers—rather than the full ε-elimination.

Example 24. One can also transform branches of any nodes in an input graph, by the combination of **l-sbl** with traversing the input graph by the structural recursion, or by multiple use of **l-sbl**. For example, the following apply transformation e to branches of all nodes pointed by a-labeled edges.

```
$apply\_lct\_a(e)(db) = \mathbf{srec}(\ \lambda(l, g).\ \mathbf{if}\ \ l = \mathtt{a}$
                    $\mathbf{then}\ \langle l : \textbf{l-sbl}(e)(g) \rangle\ @\ ()\ \ \mathbf{else}\ \langle l : \langle \& \rangle \rangle)(db)$
```

The following applies a transformation $\textbf{l-sbl}(e)$ to branches of the first branch of the root.

```
$apply\_lct\_first\_branch(e)(db) = \textbf{l-sbl}(\ \lambda bs.\mathbf{cons}($
  $\mathbf{case}\ \mathbf{car}(bs)\ \mathbf{of}\ \mathbf{Edge}(l, g) \to \mathbf{Edge}(l, \textbf{l-sbl}(e)(g))$
                 $\mathbf{or}\ y \qquad\quad \to y,$
  $\mathbf{cdr}(bs)$
  $)\ )(db)$
```

□

4.3.2 Uniform Sibling Transformation

While $\textbf{l-sbl}(f)$ transforms branches only of one node (root node), $\textbf{u-sbl}(f)$ transform branches uniformly of all nodes in a given graph, as in Figure 9.

Let us see the definition. Let

$$\phi \colon T(\mathcal{L} \times (\text{-}) + Y) \to T(\mathcal{L} \times (\text{-}) + Y)$$

be a natural transformation. For a T-graph $G = (V, B, I) \in DB_{\mathrm{f}}{}^{X}_{Y}$,

$$\textbf{u-sbl}(\phi)(G) \overset{\text{def}}{=} (V, \phi_V \circ B, I).$$

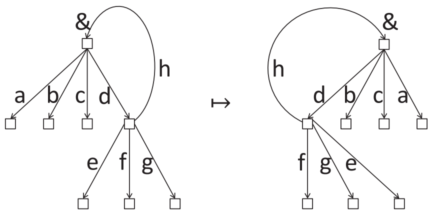


Figure 9. $\textbf{u-sbl}$(swap-hd-tl)

The naturality of ϕ implies that $\textbf{u-sbl}(\phi)$ is bisimulation generic.

In the language $\lambda^{T}_{\mathrm{FG}}$, the naturality is realized by parametricity. The term

$$\lambda x.e \colon T(\mathbf{Label} \times \alpha + Y) \to T(\mathbf{Label} \times \alpha + Y)$$

in $\textbf{u-sbl}(\lambda x.e)$ should be written as a parametric term. Then the interpretation of $\lambda x.e$ becomes a natural transformation, by the free theorem.

Functions $\textbf{u-sbl}(\phi)$ are used for transforming (a list of) branches for each node. For example, using the list reverse function $\mathbf{reverse}$, the transformation $\textbf{u-sbl}(\mathbf{reverse})$ reverses the orders of branches for all nodes in an input graph.

The introduction of **u-sbl** also enhances the expressive power of structural recursion. The structural recursion law

$$f(G +\!\!+ G') = f(G) +\!\!+ f(G')$$

with $f = \mathbf{srec}(e)$ is too restrictive in the sense that the function f cannot change the order of branches of any nodes. Under this restriction, we cannot write a transformation f which satisfies

$$f(G +\!\!+ G') = f(G') +\!\!+ f(G).$$

To circumvent the restriction, we can use the **u-sbl** construct to rearrange the order of branches during structural recursion; for example, we can write the transformation above by $\mathbf{srec}(e)(\textbf{u-sbl}(\mathbf{reverse})(G))$.

The counterexample in Example 23 is also a counterexample for the fact that if we apply $\textbf{u-sbl}(\phi)$ to graphs having ε-edges then $\textbf{u-sbl}(\phi)$ is not necessarily bisimulation generic.

If $\phi \colon T(\text{-}) \to T(\text{-})$ is not just a natural transformation but also a monad morphism such that the lifted endofunctor F_ϕ

$$F_\phi \colon \mathbf{Set}_T \to \mathbf{Set}_T$$
$$(f \colon S \to T(S')) \mapsto (\phi_{S'} \circ f \colon S \to T(S'))$$

preserves a given iteration operator, then we do not need to calculate ε-elimination before application of **u-sbl**. For example, the reverse operator $\mathbf{reverse}$ is such ϕ. We can add such *non ε-eliminating uniform sibling transformation* $\textbf{u-sbl}'(\phi)$ for ϕ satisfying the above condition, whose proof are subject to users. Or an implementer can add $\textbf{u-sbl}'(\phi)$ as primitive functions, after proving the condition for ϕ.

4.4 Syntax of $\lambda^{T}_{\mathrm{FG}}$

Finally let us see how to define the syntax of $\lambda^{T}_{\mathrm{FG}}$ for general T, modifying that of $\lambda^{List}_{\mathrm{FG}}$, i.e., Figures 4 and 5 in Section 2.2.3.

First we choose a convenient family of signature sets $\Sigma = (\Sigma(n))_{n \in \mathbb{N}}$ of T, and we replace algebraic graph constructors $[]$ and $e +\!\!+ e$ with constants $\mathbf{op}_s (s \in \Sigma)$, then we give the following typing rules.

$$\frac{\Gamma \vdash e_i \colon \mathbf{DB}^X_Y \quad (i \in n)}{\Gamma \vdash \mathbf{op}_s(e_1, ..., e_n) \colon \mathbf{DB}^X_Y} (s \in \Sigma(n))$$

Also we replace the type constructor $\mathbf{List}$ with that of the current monad T. Then we replace list operators with any convenient

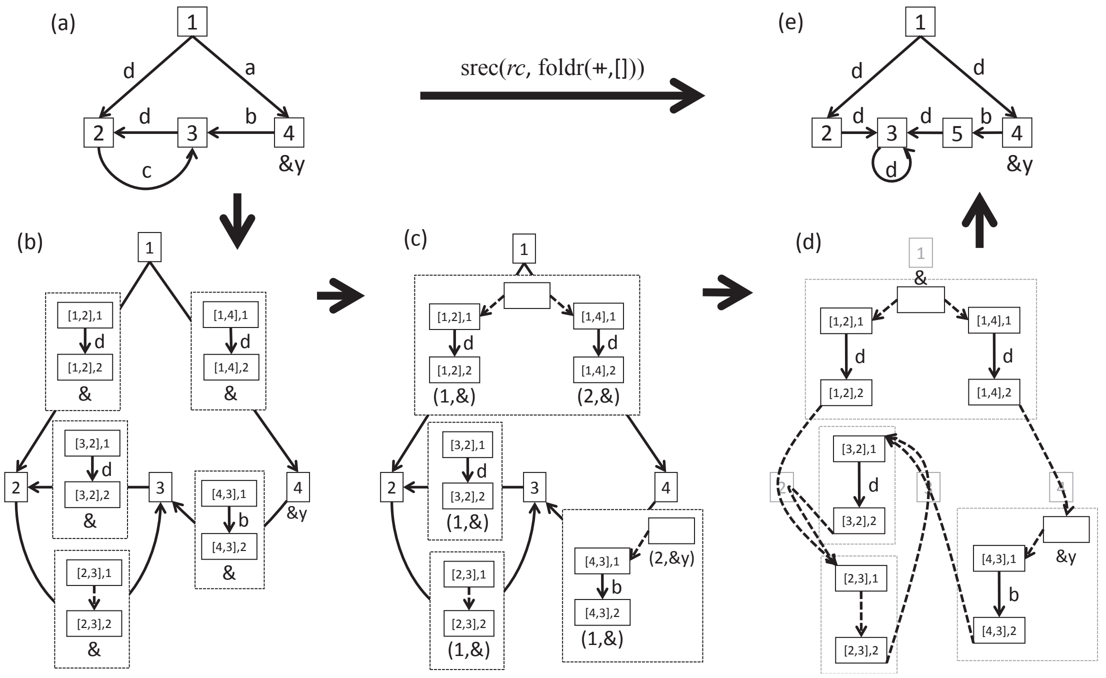


Figure 10. Bulk Semantics of Structural Recursion: An Example of $\mathbf{srec}(rc, \mathbf{foldr}(+\!\!+, []))$

operators on the monad T; according to adding operators on T, **l-sbl** and **u-sbl** become more powerful.

5. Structural Recursion with Sibling Transformation

In Section 4.3, we gave two kinds of sibling transformations. Here we show one attempt to extend the structural recursion so that by it we can transform graphs both in depth-direction and in sibling-direction at the same time. In this section we concentrate on the case when $T = List$.

First we show our idea briefly. The original structural recursion function $f = \mathbf{srec}(e)$ is characterized with the following equations:

$$f(\langle l_1 : g_1\rangle +\!\!+ \ldots +\!\!+ \langle l_n : g_n\rangle)$$
$$= e(l_1, g_1) \mathbin{@} f(g_1) +\!\!+ \ldots +\!\!+ e(l_n, g_n) \mathbin{@} f(g_n)$$

We generalize above $+\!\!+$ with arbitrary operators $\odot$ satisfying certain axioms, i.e., as the following

$$f(\langle l_1 : g_1\rangle +\!\!+ \ldots +\!\!+ \langle l_n : g_n\rangle)$$
$$= e(l_1, g_1) \mathbin{@} f(g_1) \odot \ldots \odot e(l_n, g_n) \mathbin{@} f(g_n)$$

so that we can transform graphs in sibling direction with $\odot$.

Now let us see the typing rule for the extended structural recursion function:

$$\frac{\Gamma \vdash e_1\colon \mathbf{Label}{\times}\mathbf{DB}_Y \to \mathbf{DB}^Z_Z \quad \Gamma \vdash e_2\colon \mathbf{List}(\mathbf{DB}^Z_\alpha) \to \mathbf{DB}^Z_\alpha \quad e_1 \text{ and } e_2 \text{ are production-consumption compatible}}{\Gamma \vdash \mathbf{srec}(e_1, e_2)\colon \mathbf{DB}^X_Y \to \mathbf{DB}^{Z\times X}_{Z\times Y}}$$

In the typing rule, e_2 is a parametric polymorphic term on the sets of output markers. We generalize $\odot$ above from binary operator to list operator; for a binary operator (with unit) we can construct a list operator by foldr (or foldl), where we do not need to require them to be a monoid. We explain production-consumption compatibility after giving bulk semantics of this $\mathbf{srec}$.

Before giving the formal definition of bulk semantics of $\mathbf{srec}(e, d)$, we illustrate the bulk computation behavior using $a2d\_xc \overset{\text{def}}{=} \mathbf{srec}(rc, \mathbf{foldr}(+\!\!+, []))$ (see Example 4 for rc). It consists of two steps of bulk computations followed by a grouping step, starting with the input graph (a) in Figure 10.

1. *Applying Map Computation on Edges with e*

   Apply function rc (which renames edges labeled a to those labeled d and contracts edges labeled c) to every edge labeled l and the graph g following to the edge, and yields a graph in DB^Z_Z. We call these graphs computed by $e(= rc)$ *e-graphs*; graph (b) shows e-graphs.

2. *Applying Map Computation on Nodes with d*

   For every node of the original graph, use binary operator $+\!\!+$ to combine all branching e-graphs. We call these graphs computed by $d(= \mathbf{foldr}(+\!\!+, []))$ *d-graphs*; graph (c) shows the result. The nodes 1 and 4 have more than one branches which need to be merged using $+\!\!+$.

   Before applying $d = \mathbf{foldr}(+\!\!+, [])$ on a node of the original graph, for each ith branch of the node, we add the index i to the output markers of the ith e-graph, which will be used in the next "grouping" step.

3. *Grouping Subgraphs with ε-Edges*

   Group all d-graphs computed in Step 2: for each d-graph, its output marker produced by e with an index i is connected via an ε-edge to the root of the d-graph on the node which the original ith edge points to. We do not need anymore the original nodes, and also delete indices i of the remaining output markers (e.g., 2 of $(2, \&y)$ in the graph (c)). The root of new graph is the root of the d-graph on the original root node. With these, we can get Graph (d) from Graph (c).

Graph (e) is the graph obtained by ε-elimination for Graph (d). (We can further minimize Graph (e) if necessary; then nodes $2, 3, 5$ are identified, since they are bisimilar.) We remark a difference between the two bulk semantics: the result graph (e) of $a2d\_xc \overset{\text{def}}{=} \mathbf{srec}(rc, \mathbf{foldr}(+\!\!+, []))$ is not isomorphic to the result graph (c) in Figure 2 of $a2d\_xc \overset{\text{def}}{=} \mathbf{srec}(rc)$; but on the other hand, in general $\mathbf{srec}(e)$ and $\mathbf{srec}(e, \mathbf{foldr}(+\!\!+, []))$ are bisimilar.

Now let us give a formal definition of bulk semantics of the structural recursion. For a function $f\colon X \to Y$, we define *marker-renaming graph* $\lfloor f \rfloor \overset{\text{def}}{=} \oplus_{\&x\in X} \&x := \langle f(\&x)\rangle \in DB^X_Y$.

Definition 25. For $e\colon \mathcal{L} \times DB_Y \to DB^Z_Z$, $d\colon List(DB^Z_\alpha) \to DB^Z_\alpha$, and $G = (V, B, I) \in DB^X_Y$,

$$\mathbf{srec}(e, d)(G) \overset{\text{def}}{=} (V', B', I') \in DB^{Z\times X}_{Z\times Y}$$

is defined as below.

We assume G has no ε-edges since we can take ε-elimination; if $\varepsilon\text{-elim}(G)$ has infinite width, we put an error.

We first extend e to the following dependent type function

$$\begin{aligned}&\bar{e}\colon \mathcal{L}\times V + Y \to DB^Z_{Z_x}\\ &x = \mathsf{Edge}\,(l, v) \mapsto e(l, G|_v)\ (Z_x = Z)\\ &x = \mathsf{Outm}\,(\&y) \mapsto \oplus_{\&z\in Z} \&z := \langle(\&z, \&y)\rangle\ (Z_x = Z\times Y),\end{aligned}$$

and then to the following dependent type function

$$\begin{aligned}&e_{List}\colon List(\mathcal{L}\times V + Y) \to List(DB^Z_{W_l})\\ &\qquad l = (l_i)_{i\in n} \mapsto (\bar{e}(l_i) \mathbin{@} \lfloor in_i\rfloor)_{i\in n}\ (W_l = \textstyle\coprod_{i\in n} Z_{l_i}).\end{aligned}$$

Next, for each $v \in V$, we define *d-graph on* v as

$$d_v \overset{\text{def}}{=} d(e_{List}(B(v))) \in DB^Z_{\coprod_{i=1}^n Z_i},$$

where $n = |B(v)|$ and

$$Z_i = \begin{cases} Z & (\text{if } B(v).i \text{ is an edge})\\ Z\times Y & (\text{if } B(v).i \text{ is an output marker}).\end{cases}$$

Then,

- $V' \overset{\text{def}}{=} \coprod_{v\in V} d_v.\mathrm{V}$

- $$\begin{aligned}&B'\colon \textstyle\coprod_v d_v.\mathrm{V} \to List\big(\mathcal{L}\times(\coprod_v d_v.\mathrm{V}) + Z\times Y\big)\\ &\qquad (v, u) \ \mapsto\ List(g)(\ (d_v.\mathrm{B})(u)\ )\end{aligned}$$
$$\left(\begin{array}{lll}(d_v.\mathrm{B})(u) \ \in\ List\big(\mathcal{L}\times(d_v.\mathrm{V}) + \coprod_{i=1}^n Z_i\big) & & \\ g\colon \mathcal{L}\times(d_v.\mathrm{V}) + \coprod_{i=1}^n Z_i \to \mathcal{L}\times(\coprod_v d_v.\mathrm{V}) + Z\times Y & & \\ \quad \mathsf{Edge}\,(l, u') & \mapsto & \mathsf{Edge}\,\big(l, (v, u')\big)\\ \quad \mathsf{Outm}\,((i, \&z)) & \mapsto & \text{let } B(v).i = \mathsf{Edge}\,\big(l, v'\big) \text{ in}\\ & & \mathsf{Edge}\,\big(\varepsilon, \big(v', (d_{v'}.\mathrm{I})(\&z)\big)\big)\\ \quad \mathsf{Outm}\,((i, (\&z, \&y))) & \mapsto & \mathsf{Outm}\,((\&z, \&y))\end{array}\right)$$

- $$\begin{aligned}&I'\colon Z\times X \to \textstyle\coprod_v d_v.\mathrm{V}\\ &\quad (\&z, \&x) \ \mapsto\ \big(I(\&x), (d_{I(\&x)}.\mathrm{I})(\&z)\big)\end{aligned}$$ $\square$

Now we explain the assumption in the above typing rule. Terms e_1 and e_2 are called *production-consumption compatible* if the interpretations $e = [\![e_1]\!]$ and $d = [\![e_2]\!]$ satisfy the following. Let $G_i \in DB^Z_{Z_i}, G'_i \in DB^{Z_i}_{Z'_i} (i = 1, ..., n)$ be graphs such that for each i, either

- G_i is an application of e and so $Z_i = Z$, or
- $G_i = \oplus_{\&z\in Z} \&z := \langle(\&z, \&y)\rangle$ for some $\&y \in Y$, $Z_i = Z'_i = Z\times Y$, and $G'_i = \lfloor id\rfloor$.

Then the following must be satisfied

$$\begin{aligned}&d((G_1 \mathbin{@} G'_1 \mathbin{@} \lfloor in_1\rfloor), ..., (G_n \mathbin{@} G'_n \mathbin{@} \lfloor in_n\rfloor))\\ &\quad = d(G_1 \mathbin{@} \lfloor in_1\rfloor, ..., G_n \mathbin{@} \lfloor in_n\rfloor) \mathbin{@} (G'_1 + ... + G'_n)\\ &\quad \big(\in DB^Z_{Z'_1+\dots+Z'_n}\big)\end{aligned}$$

where $G'_1 + G'_2 \overset{\text{def}}{=} (G'_1 \mathbin{@} \lfloor in_1\rfloor) \oplus (G'_2 \mathbin{@} \lfloor in_2\rfloor) \in DB^{Z_1+Z_2}_{Z'_1+Z'_2}$.

This condition means that d "consumes" only the information of the graphs "produced" by e.

By this assumption, we can show that the bulk semantics agrees with the *recursive semantics*: i.e., the structural recursion function $f = \mathbf{srec}(e, d)$ in Definition 25 satisfies (and is characterized by) the following equation

$$\begin{aligned}&f(\langle l_1 : g_1\rangle +\!\!+ \dots +\!\!+ \langle l_n : g_n\rangle)\\ &\quad = d(e(l_1, g_1) \mathbin{@} f(g_1), ..., e(l_n, g_n) \mathbin{@} f(g_n)).\end{aligned}$$

If $d = \mathbf{foldr}(\odot, \iota_\odot)$ for some $\odot$ and $\iota_\odot$, then the following characterizing equations are also available

$$\begin{array}{lcl} f([]) & = & \iota_\odot\\ f(\langle l_1 : g_1\rangle +\!\!+ g) & = & (e(l_1, g_1) \mathbin{@} f(g_1)) \odot f(g).\end{array}$$

Further, if $(\odot, \iota_\odot)$ is a monoid, then we obtain also the following simpler characterizing equations

$$\begin{array}{lcl} f([]) & = & \iota_\odot\\ f(\langle l : g\rangle) & = & e(l, g) \mathbin{@} f(g)\\ f(g_1 +\!\!+ g_2) & = & f(g_1) \odot f(g_2).\end{array}$$

Above we considered the case when X is a singleton and omitted the case when a branch might be an output marker $\&y$; for the general case, first note that

$$\mathbf{srec}(e, d)\colon DB^X_Y \to DB^{Z\times X}_{Z\times Y}$$

is bisimilar to the singleton case to the Xth power

$$\mathbf{srec}(e, d)^X\colon (DB_Y)^X \to (DB^Z_{Z\times Y})^X$$

with the isomorphisms

$$\oplus\colon (DB_Y)^X \cong DB^X_Y, \qquad \oplus\colon (DB^Z_{Z\times Y})^X \cong DB^{Z\times X}_{Z\times Y},$$

hence the case of general X is reduced to the case when X is a singleton. On the matter that a branch might be an output marker, we replace

$$e(l, g) \mathbin{@} f(g) \qquad\qquad \text{for} \quad \langle l : g\rangle$$

with

$$\oplus_{\&z\in Z} \&z := \langle(\&z, \&y)\rangle \qquad \text{for} \quad \langle \&y\rangle.$$

For example, the above second characterizing equations when $d = \mathbf{foldr}(\odot, \iota_\odot)$ becomes

$$\begin{array}{lcl} f([]) & = & \iota_\odot\\ f(\langle l_1 : g_1\rangle +\!\!+ g) & = & (e(l_1, g_1) \mathbin{@} f(g_1)) \odot f(g)\\ f(\langle \&y\rangle +\!\!+ g) & = & \oplus_{\&z\in Z} \&z := \langle(\&z, \&y)\rangle \odot f(g).\end{array}$$

Example 26. The following *left_a_path* cut down the left side of the leftmost path consisting only of a-labeled edges, as in Fig-

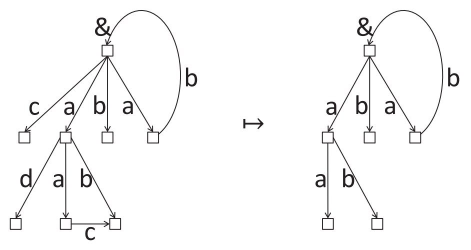


Figure 11. *left_a_path*

ure 11. We explain the detail after the definition.

$$
\begin{array}{l}
\mathit{left\_a\_path}\colon \mathbf{DB}_Y \to \mathbf{DB}_Y\\
\mathit{left\_a\_path}(g) = \langle \&\rangle \mathbin{@} f(g) \mathbin{@} \lfloor pr_r \rfloor \quad (pr_r\colon \{\&, \&z\}\times Y \to Y)\\
\quad \mathbf{where}\\
\qquad f\colon \mathbf{DB}_Y \to \mathbf{DB}^{\{\&,\&z\}}_{\{\&,\&z\}\times Y}\\
\qquad f = \mathbf{srec}(e, \mathbf{foldr}(o, i))\\
\qquad e\colon \mathbf{Label}\times\mathbf{DB}_Y \to \mathbf{DB}^{\{\&,\&z\}}_{\{\&,\&z\}}\\
\qquad e(l, g) = \langle l : \langle\&\rangle\rangle \oplus \&z := \langle l : \langle\&z\rangle\rangle\\
\qquad \mathit{is\_a\_under\_root\_of}\colon \mathbf{DB}_Y \to \mathbf{Bool}\\
\qquad \mathit{is\_a\_under\_root\_of}(g) = \mathbf{not}(\mathbf{isEmpty}(\\
\qquad\quad \mathbf{srec}(\lambda(l, g).\, \mathbf{if}\ l = \mathtt{a}\ \mathbf{then}\ \langle a : []\rangle\ \mathbf{else}\ [])(g)\ ))\\
\qquad o\colon \mathbf{DB}^{\{\&,\&z\}}_{\alpha}\times\mathbf{DB}^{\{\&,\&z\}}_{\alpha} \to \mathbf{DB}^{\{\&,\&z\}}_{\alpha}\\
\qquad o(g_1, g_2) = \mathbf{if}\ \mathit{is\_a\_under\_root\_of}(\langle\&z\rangle \mathbin{@} g_1)\\
\qquad\quad \mathbf{then}\ ((\langle\&\rangle \mathbin{@} g_1) +\!\!+ (\langle\&z\rangle \mathbin{@} g_2)) \oplus\\
\qquad\qquad\quad \&z := ((\langle\&z\rangle \mathbin{@} g_1) +\!\!+ (\langle\&z\rangle \mathbin{@} g_2))\\
\qquad\quad \mathbf{else}\ (\langle\&\rangle \mathbin{@} g_2) \oplus \&z := ((\langle\&z\rangle \mathbin{@} g_1) +\!\!+ (\langle\&z\rangle \mathbin{@} g_2))\\
\qquad i\colon \mathbf{DB}^{\{\&,\&z\}}_{\alpha}\\
\qquad i = [] \oplus \&z := []
\end{array}
$$

First, it is easily checked that the above e and $\mathbf{foldr}(o, i)$ are production-consumption compatible. In fact,

$$
\begin{array}{l}
o(e(l_1, g_1) \mathbin{@} g_1' \mathbin{@} \lfloor in_1\rfloor, e(l_2, g_2) \mathbin{@} g_2' \mathbin{@} \lfloor in_2\rfloor)\\
= \mathbf{if}\ \mathit{is\_a\_under\_root\_of}(\langle\&z\rangle \mathbin{@} e(l_1, g_1) \mathbin{@} g_1' \mathbin{@} \lfloor in_1\rfloor)\\
\quad \mathbf{then}\ ...\ \mathbf{else}\ ...\\
= \mathbf{if}\ \mathit{is\_a\_under\_root\_of}(\langle l_1 : (\langle\&z\rangle \mathbin{@} g_1' \mathbin{@} \lfloor in_1\rfloor)\rangle)\\
\quad \mathbf{then}\ ...\ \mathbf{else}\ ...\\
= \left.\begin{array}{l}
\mathbf{if}\ l_1 = a\\
\quad \mathbf{then}\ \big(((\langle\&\rangle \mathbin{@} e(l_1, g_1) \mathbin{@} g_1' \mathbin{@} \lfloor in_1\rfloor) +\!\!+\\
\qquad\quad (\langle\&z\rangle \mathbin{@} e(l_2, g_2) \mathbin{@} g_2' \mathbin{@} \lfloor in_2\rfloor)\ \big) \oplus \&z :=\\
\qquad \big(((\langle\&z\rangle \mathbin{@} e(l_1, g_1) \mathbin{@} g_1' \mathbin{@} \lfloor in_1\rfloor) +\!\!+\\
\qquad\quad (\langle\&z\rangle \mathbin{@} e(l_2, g_2) \mathbin{@} g_2' \mathbin{@} \lfloor in_2\rfloor)\ \big)\\
\quad \mathbf{else}\ \big(\langle\&\rangle \mathbin{@} e(l_2, g_2) \mathbin{@} g_2' \mathbin{@} \lfloor in_2\rfloor\big) \oplus \&z :=\\
\qquad \big(((\langle\&z\rangle \mathbin{@} e(l_1, g_1) \mathbin{@} g_1' \mathbin{@} \lfloor in_1\rfloor) +\!\!+\\
\qquad\quad (\langle\&z\rangle \mathbin{@} e(l_2, g_2) \mathbin{@} g_2' \mathbin{@} \lfloor in_2\rfloor)\ \big)
\end{array}\right\}(*)
\end{array}
$$

and on the other hand,

$$
\begin{array}{l}
o(e(l_1, g_1) \mathbin{@} \lfloor in_1\rfloor, e(l_2, g_2) \mathbin{@} \lfloor in_2\rfloor) \mathbin{@} (g_1' + g_2')\\
= \dots\\
= \big(\ \mathbf{if}\ l_1 = a\\
\quad \mathbf{then}\ \big(((\langle\&\rangle \mathbin{@} e(l_1, g_1) \mathbin{@} \lfloor in_1\rfloor) +\!\!+\\
\qquad\quad (\langle\&z\rangle \mathbin{@} e(l_2, g_2) \mathbin{@} \lfloor in_2\rfloor)\ \big) \oplus \&z :=\\
\qquad \big(((\langle\&z\rangle \mathbin{@} e(l_1, g_1) \mathbin{@} \lfloor in_1\rfloor) +\!\!+\\
\qquad\quad (\langle\&z\rangle \mathbin{@} e(l_2, g_2) \mathbin{@} \lfloor in_2\rfloor)\ \big)\\
\quad \mathbf{else}\ \big(\langle\&\rangle \mathbin{@} e(l_2, g_2) \mathbin{@} \lfloor in_2\rfloor\big) \oplus \&z :=\\
\qquad \big(((\langle\&z\rangle \mathbin{@} e(l_1, g_1) \mathbin{@} \lfloor in_1\rfloor) +\!\!+\\
\qquad\quad (\langle\&z\rangle \mathbin{@} e(l_2, g_2) \mathbin{@} \lfloor in_2\rfloor)\ \big)\\
\ \big) \mathbin{@} (g_1' + g_2')\\
= (*).
\end{array}
$$

The $\&z$-root graph of $f(g)$ is the original graph g; in fact, by the recursive semantics, we can check that $\langle\&z\rangle \mathbin{@} f(g) \mathbin{@} \lfloor pr_r\rfloor = g$ as below:

$$\langle\&z\rangle \mathbin{@} f([]) = \langle\&z\rangle \mathbin{@} i = [],$$

$$
\begin{array}{l}
\langle\&z\rangle \mathbin{@} f(\langle l_1 : g_1\rangle +\!\!+ g)\\
= \langle\&z\rangle \mathbin{@} o(e(l_1, g_1) \mathbin{@} f(g_1), f(g))\\
= \mathbf{if}\ \mathit{is\_a\_under\_root\_of}(\langle\&z\rangle \mathbin{@} e(l_1, g_1) \mathbin{@} f(g_1))\\
\quad \mathbf{then}\ \langle\&z\rangle \mathbin{@} e(l_1, g_1) \mathbin{@} f(g_1) +\!\!+ (\langle\&z\rangle \mathbin{@} f(g))\\
\quad \mathbf{else}\ \langle\&z\rangle \mathbin{@} e(l_1, g_1) \mathbin{@} f(g_1) +\!\!+ (\langle\&z\rangle \mathbin{@} f(g))\\
= \langle\&z\rangle \mathbin{@} e(l_1, g_1) \mathbin{@} f(g_1) +\!\!+ (\langle\&z\rangle \mathbin{@} f(g))\\
= \langle l_1 : (\langle\&z\rangle \mathbin{@} f(g_1))\rangle +\!\!+ (\langle\&z\rangle \mathbin{@} f(g)).
\end{array}
$$

(The above is paramorphism of structural recursion by tupling, regarding multi-rootedness as tuple.)

Then let us see $\mathit{left\_a\_path}(g)$, i.e., the $\&$-root graph of $f(g)$:

$$\langle\&\rangle \mathbin{@} f([]) = \langle\&\rangle \mathbin{@} i = [],$$

$$
\begin{array}{l}
\langle\&\rangle \mathbin{@} f(\langle l_1 : g_1\rangle +\!\!+ g)\\
= \langle\&\rangle \mathbin{@} o(e(l_1, g_1) \mathbin{@} f(g_1), f(g))\\
= \mathbf{if}\ \mathit{is\_a\_under\_root\_of}(\langle\&z\rangle \mathbin{@} e(l_1, g_1) \mathbin{@} f(g_1))\\
\quad \mathbf{then}\ (\langle\&\rangle \mathbin{@} e(l_1, g_1) \mathbin{@} f(g_1)) +\!\!+ (\langle\&z\rangle \mathbin{@} f(g))\\
\quad \mathbf{else}\ (\langle\&\rangle \mathbin{@} f(g))\\
= \mathbf{if}\ \mathit{is\_a\_under\_root\_of}(\langle l_1 : (\langle\&z\rangle \mathbin{@} f(g_1))\rangle)\\
\quad \mathbf{then}\ \langle l_1 : (\langle\&\rangle \mathbin{@} f(g_1))\rangle +\!\!+ (\langle\&z\rangle \mathbin{@} f(g))\\
\quad \mathbf{else}\ (\langle\&\rangle \mathbin{@} f(g))\\
= \mathbf{if}\ \mathit{is\_a\_under\_root\_of}(\langle l_1 : g_1\rangle)\\
\quad \mathbf{then}\ \langle l_1 : (\langle\&\rangle \mathbin{@} f(g_1))\rangle +\!\!+ g\\
\quad \mathbf{else}\ (\langle\&\rangle \mathbin{@} f(g)).
\end{array}
$$

Now it is clear that the function $\mathit{left\_a\_path}(= \langle\&\rangle \mathbin{@} f(\text{-}))$ works as in Figure 11. □

By the bulk semantics, it is clear that the above structural recursion function preserves finiteness of graphs. Also, similarly to Theorem 22, we can show that the above structural recursion is bisimulation generic.

Comparison of Sibling Transformations

The structural recursion extended above is similar to the combination of **u-sbl** and the original structural recursion in Section 4.2. in fact, when $e\colon \mathcal{L}\times DB_Y \to DB_{\{\&\}}$ is "identity" $i(l, g) \overset{\text{def}}{=} \langle l : g\rangle$, $\mathbf{srec}(i, d)$ is similar to $\mathbf{u\text{-}sbl}(f)$, though here d and f have a bit different types. The main difference between the extended structural recursion and the combination of **u-sbl** and the original structural recursion is the following. In general, $\mathbf{srec}(e, d)$ transforms graphs in sibling direction by traversing graphs produced by e. This sibling transformation can not be simulated by $\mathbf{u\text{-}sbl}(f)$ used after the original $\mathbf{srec}$; in the latter f traverses just one step of children, while in the former d can traverse more deep descendants as long as production-consumption compatibility holds.

The extended structural recursion includes also the feature of **l-sbl** in the following sense. Let

$$f\colon \mathbf{List}(\mathbf{Label}\times\mathbf{DB}_0{+}0) \to \mathbf{List}(\mathbf{Label}\times\mathbf{DB}_0{+}0)$$

be a function such that there is

$$f'\colon \mathbf{List}(\mathbf{DB}_\alpha^{\{\&,\&z\}}) \to \mathbf{DB}_\alpha$$

satisfying

$$\begin{aligned} f' \circ List(pair) &= uf^{-1} \circ f \\ &\big(= \mathbf{foldr}(+\!\!+, []) \circ List([\langle\text{-} : \text{-}\rangle, \langle\text{-}\rangle]) \circ f\ \big) \end{aligned}$$

where

$$\begin{aligned} &pair\colon \mathbf{Label}\times\mathbf{DB}_0 \to \mathbf{DB}_0^{\{\&,\&z\}} \\ &pair(l,g) \overset{\text{def}}{=} \langle l : []\rangle \oplus \&z := g. \end{aligned}$$

Then $\mathbf{l\text{-}sbl}(f)$ can be represented by $\mathbf{srec}$ as the following:

$$\mathbf{l\text{-}sbl}(f) = \langle\&\rangle \mathbin{@} \mathbf{srec}(e,d)$$

where

$$\begin{aligned} &e\colon \mathbf{Label}\times\mathbf{DB}_0 \to \mathbf{DB}_{\{\&,\&z\}}^{\{\&,\&z\}} \\ &e(l,g) \overset{\text{def}}{=} \langle l : []\rangle \oplus \&z := (g \mathbin{@} \lfloor !\rfloor) \quad (!\colon 0 \to \{\&,\&z\}) \\ &d\colon \mathbf{List}(\mathbf{DB}_\alpha^{\{\&,\&z\}}) \to \mathbf{DB}_\alpha^{\{\&,\&z\}} \\ &d(l) \overset{\text{def}}{=} f'(l) \oplus \&z := []\,. \end{aligned}$$

(Here again we used tupling technique for $\mathbf{srec}$:) This is because,

$$\begin{aligned} &\mathbf{srec}(e,d)(\langle l_1 : g_1\rangle +\!\!+ \ldots +\!\!+ \langle l_n : g_n\rangle) \\ &= d(e(l_1,g_1) \mathbin{@} \mathbf{srec}(e,d)(g_1), \ldots, e(l_n,g_n) \mathbin{@} \mathbf{srec}(e,d)(g_n)) \\ &= d(pair(l_1,g_1), \ldots, pair(l_n,g_n)) \end{aligned}$$

hence,

$$\begin{aligned} &\langle\&\rangle \mathbin{@} \mathbf{srec}(e,d)(\langle l_1 : g_1\rangle +\!\!+ \ldots +\!\!+ \langle l_n : g_n\rangle) \\ &= f'(pair(l_1,g_1), \ldots, pair(l_n,g_n)) \\ &= (f' \circ List(pair))((l_1,g_1), \ldots, (l_n,g_n)) \\ &= (uf^{-1} \circ f)(uf(\langle l_1 : g_1\rangle +\!\!+ \ldots +\!\!+ \langle l_n : g_n\rangle)) \\ &= \mathbf{l\text{-}sbl}(f)(\langle l_1 : g_1\rangle +\!\!+ \ldots +\!\!+ \langle l_n : g_n\rangle). \end{aligned}$$

6. Modularized Extension of UnCAL

In Section 4, we saw how we can generalize (and extend) UnCAL to obtain λ_{FG}^T with monads T which satisfy certain assumptions: i.e.,

- T has an extension for ε-elimination,
- T preserves weak-pullbacks, and
- T is a finitary monad.

Also we saw four examples of such monads: P_{fin}, $List$, M_{fin}, and D_{fin}. Here we see how we can compose monads T to obtain λ_{FG}^T.

Since monad was introduced for the notion of computational effects [Moggi 1989], there has been much study about how to compose monads from smaller monads: i.e., monad transformers. In [Benton et al. 2000] many examples of unary monad transformers are listed. In [Hyland et al. 2006], the authors studied binary monad transformers and show also how they can be used to produce unary monad transformers such as those in the former paper.

In the following, as a demonstration of our modular approach, we take up one simple binary monad transformer—the product $T{\times}T'$ of monads—, and see the language $\lambda_{\mathrm{FG}}^{T\times T'}$.

6.1 Product of Monads

For a pair of monads T_1 and T_2 on **Set**, we define their product $T_1{\times}T_2$ just as

$$(T_1{\times}T_2)(X) \overset{\text{def}}{=} T_1(X){\times}T_2(X)$$

for a set X and

$$(T_1{\times}T_2)(f) \overset{\text{def}}{=} T_1(f){\times}T_2(f)$$

for a function f. Then

$$return^{T_1\times T_2} \overset{\text{def}}{=} \big\langle return^{T_1}, return^{T_2}\big\rangle\colon X \to (T_1{\times}T_2)(X),$$

and for $f\colon X \to T_1(Y){\times}T_2(Y)$,

$$lift^{T_1\times T_2}(f) \overset{\text{def}}{=} lift^{T_1}(pr_l \circ f) \times lift^{T_2}(pr_r \circ f).$$

From now we check that taking the product of monads preserves the above three assumptions.

It is easy to check that if two monads T_1 and T_2 have uniform iteration operators in their Kleisli categories, so does the product $T_1{\times}T_2$: for $f\colon X \to T_1(X{+}Y){\times}T_2(X{+}Y)$,

$$\begin{aligned} &iter^{T_1\times T_2}(f) \overset{\text{def}}{=} \\ &\qquad \big\langle iter^{T_1}(pr_l \circ f), iter^{T_2}(pr_r \circ f)\big\rangle\colon X \to T_1(Y){\times}T_2(Y). \end{aligned}$$

If monads T_1 and T_2 preserve weak pullbacks, so does $T_1{\times}T_2$, since the product functor $\times$ preserve them.

Also it is clear on finitarity since the product functor is finitary, but let us see concretely what family of signature sets for the product monad $T_1{\times}T_2$ we can get from families of signature sets for T_1 and T_2 since it is needed to define syntax and for implementation.

Let Σ_1 and Σ_2 be families of signature sets for T_1 and T_2, respectively. Then $(\Sigma_1{\otimes}\Sigma_2)(n)$ is defined as the following:

$$\begin{aligned} &\{(f_1,f_2) \in T_1(n){\times}T_2(n) \mid m_1,m_2 \in \mathbb{N},\ n{=}\max(m_1,m_2), \\ &\qquad\qquad f_1 \in \Sigma_1(m_1)(\subseteq T_1(m_1) \subseteq T_1(n)), \\ &\qquad\qquad f_2 \in \Sigma_2(m_2)(\subseteq T_2(m_2) \subseteq T_2(n))\ \} \\ &+ \{(f_1, return(n{-}1)) \in T_1(n){\times}T_2(n) \mid \\ &\qquad\qquad n > 0,\ f_1 \in \Sigma_1(n{-}1)(\subseteq T_1(n{-}1))\} \\ &+ \{(return(n{-}1), f_2) \in T_1(n){\times}T_2(n) \mid \\ &\qquad\qquad n > 0,\ f_2 \in \Sigma_2(n{-}1)(\subseteq T_2(n{-}1))\} \\ &+ \{(0,1) \in T_1(n){\times}T_2(n) \mid n = 2\ \} \end{aligned}$$

where recall that we regard a natural number n as the set $\{0,\ldots,n{-}1\}$. Then one can check that this $\Sigma_1{\otimes}\Sigma_2$ becomes a family of signature sets for $T_1{\times}T_2$ by induction.

Example 27. Let us consider the case when $T_1 = P_{\mathrm{fin}}$ and $T_2 = List$.

Now we have two signatures $\{\}$ and $\{0,1\}$—which correspond to the syntax $\{\}$ and $\cup$, respectively—for P_{fin}, and also two signatures $[]$ and $[0,1]$—which correspond to the syntax $[]$ and $+\!\!+$, respectively—for $List$, hence we should have nine $(= 2{\times}2{+}2{+}2{+}1)$ signatures for $P_{\mathrm{fin}}{\times}List$. However some of them can be represented as compositions of the other ones and common graph constructors, then after all signatures we need for $P_{\mathrm{fin}}{\times}List$ are: $(\{\},[])$ (nullary), $(\{0,1\},[0,1])$ (binary), $(\{\}, return(0))$ (unary), and $(return(0),[])$ (unary).

Let the syntax corresponding to these four signatures be

$$(\{\},[]) \mid e(\cup,+\!\!+)e \mid \mathbf{del\text{-}unorder}(e) \mid \mathbf{del\text{-}order}(e)$$

and replace the part $\{\text{algebraic graph operators}\}$ of syntax of $\lambda_{\mathrm{FG}}^{List}$ in Figure 4 with these four. Also replace all the occurrences of type $\mathbf{List}(...)$ with $\mathbf{Set}(...)\times\mathbf{List}(...)$ both in the BNF type definition of $\lambda_{\mathrm{FG}}^{List}$ and in the typing rules. The resulting syntax is the core syntax of $\lambda_{\mathrm{FG}}^{P_{\mathrm{fin}}\times List}$.

The nullary operator $(\{\}, [])$ produces single node graph with no branches. The binary operator $(\cup, +\!\!+)$ takes union of unordered branches of the roots of the two argument graphs, and at the same time, takes append of ordered branches of the roots of the two argument graphs. The unary operator $\mathbf{del\text{-}unorder}(e)$ deletes all unordered branches only of the roots of an argument graph, so e.g. unordered branches of ordered branches of the root remain. Similarly, the unary operator $\mathbf{del\text{-}unorder}(e)$ deletes all ordered branches only of the roots of an argument graph.

As we can add any list operator such as $\mathbf{foldr}$ or $\mathbf{filter}$, we can add any operator for $P_{\mathrm{fin}}\times List$ to the core syntax of $\lambda_{\mathrm{FG}}^{P_{\mathrm{fin}}\times List}$.

For example, we can add the function $\mathbf{l2s}$ which maps lists to finite sets by forgetting the orders of lists, then we can define the term

$$x\colon \mathbf{Set}(b)\times\mathbf{List}(b) \vdash (\pi^{\mathbf{l}}(x)\cup\mathbf{l2s}(\pi^{\mathbf{r}}(x)), [])\colon \mathbf{Set}(b)\times\mathbf{List}(b)$$

and apply $\mathbf{u\text{-}sbl}$ to this to get a function which transform all ordered branches to unordered ones.

7. Related Work

Structural recursion for graphs, which we generalized, is much related to research on algebras of programming [Bird and de Moor 1996; Gibbons 2002; Hu et al. 2006; Meijer et al. 1991], where structural recursion such as folds and catamorphisms are used to structure programs and to systematically manipulate programs. In particular, our approach is influenced by many attempts of defining structural recursion for various kinds of specific graphs, such as directed acyclic graphs [Gibbons 1995], graphs represented by trees with specific pointers [Dal Zilio et al. 2004; Hamana 2009], and graphs represented by trees with embedded functions [Fegaras and Sheard 1996]. However, all these attempts are not easy to be applied in practice, due to lack of expressive power or difficulty in guaranteeing finite representation of well-formed graphs with no dangling pointers.

As described in the introduction, it is the structural recursion in UnCAL [Buneman et al. 2000] that is more practical for manipulating unordered graphs. This forms the basis of our work. Our $\lambda_{\mathrm{FG}}^{T}$ is parameterized with monad T, and the case $T = P_{\mathrm{fin}}$, i.e., $\lambda_{\mathrm{FG}}^{P_{\mathrm{fin}}}$ is comparable with UnCAL. The following is the comparison between UnCAL and $\lambda_{\mathrm{FG}}^{P_{\mathrm{fin}}}$. The two graph models are equivalent up to each bisimilarity, which for UnCAL is called value equivalence in [Buneman et al. 2000]. Also the graph constructors and structural recursion are the same. Then we extended expressive power in two points. One is the extension on sibling transformations: i.e., $\mathbf{l\text{-}sbl}$ and $\mathbf{u\text{-}sbl}$. The other is the extension of type system; UnCAL is first order calculus, while $\lambda_{\mathrm{FG}}^{P_{\mathrm{fin}}}$ is a simply typed lambda calculus (extended with graph types). This extension of type system is not for free, as we noted before Theorem 22.

A lot of work has been devoted to efficient implementation of graph algorithms in lazy functional languages [Burton and Yang 1990; Erwig 1997; Johnsson 1998; King and Launchbury 1995]. The emphasis there is placed on the importance of achieving efficient implementation of general graph algorithms through the monadic model for including actions on the state in the non-strict context. In contrast, we focus on inductive traversals of ordered graphs and aim to provide an efficient way to deal with a specific class of important graph algorithms – graph querying. Erwig shows that active patterns can be used to implement an inductive view of graphs [Erwig 2001], but that inductive view is indirect in that the graph view is dynamically maintained.

In coalgebra theory, which studies infinitary/cyclic structure, some work focused also on finiteness of graphs, as in the current paper.

In the work [Bonsangue et al. 2009], for every Kripke polynomial endofunctor F, a systematic way for giving a syntax fully representing all finite F-coalgebras and a sound and complete equational theory for bisimilarity was given. The differences between this work and our work are (i) the two classes of endofunctors—Kripke polynomial ones and ours with monads—are incomparable, (ii) the work does not treat ε-edge, and is not a study for transformation (iii) in our work the equational theory is restrictive, and (iv) the approaches to give each syntax are different.

In the work [Adámek et al. 2006; Milius 2010], the authors studied categorical properties of the set of finite coalgebras, and characterized it as the *final locally finite coalgebra*. The class of endofunctors for coalgebra in this work is wider than ours; some of the results are applied or influence to our work. In this work, there is no consideration for finiteness-preserving structural recursion. The finality among locally finite coalgebras is a kind of corecursion, and has the similar problem to that of corecursion; i.e., to assure finiteness-preservation, we have to check locally finiteness of infinite graphs, automation of which seems difficult.

A study of general framework for bisimilarity involving ε-edge was implicitly started in the paper [Jacobs 2010a]. They gave some sufficient condition for a monad to have an iteration operator in the Kleisli category. However, the theory can not be applied to the case of $CList$, which does not satisfy the sufficient condition. So the current paper gave a new example of iteration operator in a Kleisli category and of such "ε-elimination as iteration operator" perspective.

8. Conclusions and Future Work

In this paper, we present the first solution to the open problem of how to modify the graph model and structural recursion from unordered graphs to ordered ones, based on which we define a new graph transformation language $\lambda_{\mathrm{FG}}^{List}$. The key technical contributions here is the definition of bisimulation relation on ordered graphs having ε-edges. We also extend expressive power on sibling dimension with two new operators: local sibling transformation and uniform sibling transformation.

Furthermore, we generalize these results for ordered graphs with monads with suitable assumptions, and propose a more general graph languages $\lambda_{\mathrm{FG}}^{T}$ which are parameterized by monads T. This abstraction by monad T enables us to compose a language targeting bigger graph model from smaller graph models by monad transformers. We demonstrate it with the product monad transformer.

There are many interesting and important future extensions. First, we have discussed little about analysis of structural recursion. One interesting analysis is when a structural recursion function is productive. A graph function is said to be productive, if it produces a finite ordered graph without ε-edges for any input ordered graph without ε-edge. Second, following our previous work of bidirectionalizing UnCAL [Hidaka et al. 2010], we are very interested in a systematic way to bidirectionalize $\lambda_{\mathrm{FG}}^{T}$, which is indeed the first motivation of this work.

Acknowledgments Thanks are due to Ichiro Hasuo and Naohiko Hoshino for helpful discussions.

References

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

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.

R. Bird and O. de Moor. *Algebras of Programming*. Prentice Hall, 1996.

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

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

F. W. Burton and H.-K. Yang. Manipulating multilinked data structures in a pure functional language. *Softw. Pract. Exper.*, 20:1167–1185, November 1990.

S. Dal Zilio, D. Lugiez, and C. Meyssonnier. A logic you can count on. In *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL'04, pages 135–146, New York, NY, USA, 2004. ACM.

M. Erwig. Functional programming with graphs. In *Proceedings of the second ACM SIGPLAN international conference on Functional programming*, ICFP '97, pages 52–65, New York, NY, USA, 1997. ACM.

M. Erwig. Inductive graphs and functional graph algorithms. *J. Funct. Program.*, 11:467–492, September 2001. ISSN 0956-7968.

L. Fegaras and T. Sheard. Revisiting catamorphisms over datatypes with embedded functions. In *Proc. ACM symposium on principles of programming languages*, St. Petersburg Beach, Florida, Jan. 1996.

J. Gibbons. An initial-algebra approach to directed acyclic graphs. In *Mathematics of Program Construction*, MPC '95, pages 282–303, London, UK, 1995. Springer-Verlag.

J. Gibbons. *Calculating functional programs*, pages 149–201. Springer-Verlag New York, Inc., New York, NY, USA, 2002.

A. Gill, J. Launchbury, and S. P. Jones. A short cut to deforestation. In *Proc. Conference on Functional Programming Languages and Computer Architecture*, pages 223–232, Copenhagen, June 1993.

S. Ginali. Regular trees and the free iterative theory. *J. Comput. Syst. Sci.*, 18(3):228–242, 1979.

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

M. Hamana. Initial algebra semantics for cyclic sharing structures. In *Proceedings of the 9th International Conference on Typed Lambda Calculi and Applications*, TLCA '09, pages 127–141, Berlin, Heidelberg, 2009. Springer-Verlag.

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

I. Hasuo, 2011. personal communication.

S. Hidaka, Z. Hu, K. Inaba, H. Kato, K. Matsuda, and K. Nakano. Bidirectionalizing graph transformations. In *ACM SIGPLAN International Conference on Functional Programming*, pages 205–216. ACM, 2010.

Z. Hu, T. Yokoyama, and M. Takeichi. Program optimizations and transformations in calculational form. In *Summer School on Generative and Transformational Techniques in Software Engineering*, pages 139–164, Braga, Portugal, 2006. Springer, LNCS 4043.

M. Hyland, G. Plotkin, and J. Power. Combining effects: sum and tensor. *Theor. Comput. Sci.*, 357(1):70–99, July 2006. ISSN 0304-3975.

B. Jacobs. From coalgebraic to monoidal traces. *Electronic Notes in Theoretical Computer Science*, 264(2):125 – 140, 2010a. Proceedings of the Tenth Workshop on Coalgebraic Methods in Computer Science (CMCS 2010).

B. Jacobs. From coalgebraic to monoidal traces. *Electron. Notes Theor. Comput. Sci.*, 264:125–140, August 2010b. ISSN 1571-0661.

T. Johnsson. Efficient graph algorithms using lazy monolithic arrays. *J. Funct. Program.*, 8:323–333, July 1998.

F. Jouault and J. Bézivin. KM3: A DSL for metamodel specification. In *Formal Methods for Open Object-Based Distributed Systems*, pages 171–185. LNCS 4037, Springer, 2006.

Y. Kakutani. Duality between call-by-name recursion and call-by-value iteration. In J. C. Bradfield, editor, *CSL*, volume 2471 of *Lecture Notes in Computer Science*, pages 506–521. Springer, 2002. ISBN 3-540-44240-5.

D. J. King and J. Launchbury. Structuring depth-first search algorithms in haskell. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '95, pages 344–354, New York, NY, USA, 1995. ACM.

E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Proc. Conference on Functional Programming Languages and Computer Architecture* (LNCS 523), pages 124–144, Cambridge, Massachuetts, Aug. 1991.

S. Milius. A sound and complete calculus for finite stream circuits. *Logic in Computer Science, Symposium on*, 0:421–430, 2010.

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

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

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

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.

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

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

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

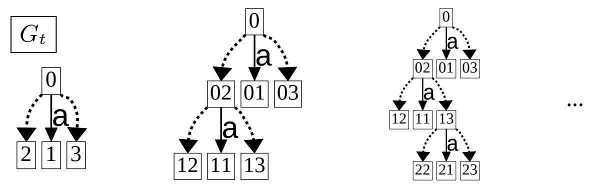


Figure 12. Branch with Any Countable Linear Order

A. Ordered Graphs for Any Countable Linear Ordered Sets

We see here that, for any countable linear ordered set, there is an infinite ordered graph having ε-edges such that the branches of the root after eliminating ε-edges are exactly as the given ordered set.

Now let L be an infinite countable linear ordered set, so there exist a bijective function $f\colon \mathbb{N} \overset{\cong}{\to} L$. Then we construct a graph "representing L" as below. Let us consider the kind of ternary tree G_t as the left tree in Figure 12. We call a leaf in a tree *ε-leaf* if the path from the root to the leaf consists of just ε-edges. First, corresponding to $f(0) \in L$, we take a fresh G_t. Next, we again take a fresh G_t, and if $f(1) < f(0)$, then we put together the root of the new G_t and the left ε-leaf of the former graph, as the middle graph in Figure 12. Otherwise if $f(0) < f(1)$, we put similarly to the right ε-leaf (03 in the example). Next, now there are three ε-leaves (12, 13, 03 in the example) with the obvious linear order, i.e., $12 < 13 < 03$. Then according to the three possible positions of $f(2)$ in the linear ordered set $\{f(0), f(1), f(2)\}$, we again put new fresh G_t to the corresponding position of ε-leaf. The right graph in Figure 12 is the example of the case when $f(1) < f(2) < f(0)$. Iterating as this, we have an infinite tree with finite (in fact just three) width. Then it is obvious that the countable linear ordered set occurring in the result graph of ε-elimination applied to this tree is order isomorphic to L.

B. Countable List

Here we give a detail explanation about the countable list monad $CList$.

First we give a formal (and abstract) definition: $CList(S)$ is the object-set of the skeleton category 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.

Now let us unfold this definition. An object of the comma category $(U \downarrow S)$ is a pair (L, l) of a countable linear ordered set L and a function $l\colon L \to S$. A morphism f from (L, l) to (L', l') is a monotone function $f\colon L \to L'$ such that $l' \circ f = l$. Then we can take the object-set $CList(S)$ of a *skeleton* category of $(U \downarrow S)$, i.e., we have a subset $CList(S)$ of the object-set of $(U \downarrow S)$ such that for any object (L, l) in $(U \downarrow S)$, there is the unique $R(L, l) \in CList(S)$ such that (L, l) and $R(L, l)$ are isomorphic in $(U \downarrow S)$, and then also there is a chosen isomorphism $\eta_{(L,l)}\colon (L, l) \to R(L, l)$.

There is another equivalent form to $CList(S)$. Let $\mathbb{L}$ be the object-set of a skeleton category of $\mathbf{CLO}$, i.e., we have a subset $\mathbb{L}$ of the object-set of $\mathbf{CLO}$ such that for any object L in $\mathbf{CLO}$, there is the unique $R(L) \in \mathbb{L}$ such that L and $R(L)$ are isomorphic in $\mathbf{CLO}$, and then also there is a chosen isomorphism $\eta_L\colon L \to R(L)$.

Then, we give the following isomorphism

$$CList(S) \cong \Sigma_{L\in\mathbb{L}}(S^L/_{\cong})$$

where the $\cong$ in the right hand side, is the restriction of the isomorphism equivalence on $|(U \downarrow S)|$ to its subset S^L; i.e., for $l, l'\colon L \to S$, $l \cong l'$ if there is an isomorphism $f\colon L \to L$ such that $l' \circ f = l$. The above correspondence is given as the following: for $(L, l) \in CList(S)$,

$$(R(L), [l \circ \eta_L^{-1}]_{\cong}) \in \Sigma_{L\in\mathbb{L}}(S^L/_{\cong}),$$

and for $(L, [l]) \in \Sigma_{L\in\mathbb{L}}(S^L/_{\cong})$,

$$R(L, l) \in CList(S),$$

which is well-defined, i.e., if (L, l) and (L, l') are isomorphic, then $R(L, l)$ and $R(L, l')$ are equal by the uniqueness in the definition of R. It is easily checked that this correspondence is bijective.

In the body text, we omit the taking quotient in the right hand side of the above isomorphism for simplicity.

C. Finitary Monad Extension for ε-elimination

Let T be a monad and $iter$ be an uniform iteration operator in $\mathbf{Set}_T$. We define ε-elimination for finite $T|_{\mathrm{fin}}$-graphs, as the result graph is equal to one defined by Definition 9.

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

$$\begin{tikzcd}
V \arrow[r, "B"] \arrow[rrd, "B'"'] & T|_{\mathrm{fin}}(\mathcal{L}_\epsilon{\times}V{+}Y) \arrow[r, "\cong"] & T|_{\mathrm{fin}}(V{+}(\mathcal{L}{\times}V{+}Y)) \\
 & & T|_{\mathrm{fin}}(V{+}(\mathcal{L}'{\times}V{+}Y)) \arrow[u, hook]
\end{tikzcd}$$

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 therefore can apply the iteration operator $iter$ to B'. We define B'' as the composition of the below

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

and then define $\varepsilon\text{-elim}(G) \overset{\mathrm{def}}{=} (V, B'', I)$.

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. Then it can be easily checked that the above $\varepsilon\text{-elim}(G)$ is exactly equal to $\varepsilon\text{-elim}(G)$ defined in Definition 9 for the T-graph G.

D. Proof of Full Representability by the Graph Constructors

Here we give two proofs of Proposition 19: full representability of finite graphs by the graph constructors. The first one is simpler and highly depends on the notion of marker, while the second one is in a naive way, which less depends on the notion of markers. The second proof shows that the notion of marker is—very convenient as in the first proof but—not essential for the fact of full representability.

First let us see the notion of 1-step unfolding uf_ε, which is almost the same as that of uf defined in Section 4.3.1. Here we consider graphs having ε-edges, while in Section 4.3.1 we consider graphs having no ε-edge. However the two are essentially the same notion since here we first consider strong bisimilarity for properties of uf_ε itself, and then we move to the level of bisimilarity by the fact that strong bisimilarity implies bisimilarity.

Now we define the 1-step unfolding

$$uf_\varepsilon\colon DB_{\mathrm{f}Y} \to T(\mathcal{L}_\epsilon{\times}DB_{\mathrm{f}Y}{+}Y).$$

First, for a graph $G = (V, B, I) \in DB^X_Y$, and a node $v \in V$, we define

$$G|_v \overset{\mathrm{def}}{=} (V, B, \{\texttt{\&} \mapsto v\}) \in DB_Y.$$

Then for $G = (V, B, I) \in DB_{\mathrm{f}Y}$,

$$uf_\varepsilon(G) \overset{\mathrm{def}}{=} T(\mathcal{L}_\epsilon{\times}f{+}Y)(B(I(\texttt{\&})))$$

where $f \overset{\text{def}}{=} G|_{(\text{-})} \colon V \to DB_{\mathrm{f}\,Y}$.

By the results [Adámek et al. 2006, Theorem 3.3] with [Milius 2010, Corollary III.15], we can show that for any finitary monad T, this uf_{ε} is isomorphic up to strong bisimilarity.

Furthermore, $uf_{\varepsilon}{}^{\text{-1}}$ is constructed just by graph constructors:

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

where

$$t \colon T(DB_{\mathrm{f}\,Y}) \to DB_{\mathrm{f}\,Y}$$

is defined with the T-algebraic graph constructors using the finitarity $T(S) = \bigcup_{i \in \mathbb{N}} T_{\Sigma}^{(i)}(S)$, and

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

For example, when $T = \mathit{List}$,

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

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 $\lambda_{\mathrm{FG}}^{T}$ 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 see the first proof.

Proof. Let $G = (V, B, I)$ be a finite graph in $DB_{\mathrm{f}\,Y}^{X}$. 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,

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

can be represented by graph constructors. Then

$$\lfloor f \circ I \rfloor \mathbin{@} \mathbf{cycle}(\oplus_{v \in V} \&v := G_v) \in DB_{Y}^{X}$$

is bisimilar to the original graph G.

For example, for the graph G in Example 2,

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

then G is bisimilar to

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

□

Now we see the second proof. First we define "node in cycle". For a finite graph (V, B, I) and $v \in V$ we define P_v as the minimum subset $P \subseteq V$ such that $B(v) \in T(\mathcal{L}_{\epsilon} \times P + Y)$; this is the set of all the nodes which are targets of edges from v. Then we define $P_v^0 \overset{\text{def}}{=} \{v\}$ and $P_v^{n+1} \overset{\text{def}}{=} \bigcup_{v' \in P_v^n} P_{v'}$ by induction on n. Now we define $P_v^{*} \overset{\text{def}}{=} \bigcup_{n \in N} P_v^n$ and $P_v^{+} \overset{\text{def}}{=} \bigcup_{n > 0} P_v^n$; P_v^{*} is the set of the nodes which are accessible from v, and P_v^{+} is the set of the nodes which are accessible from v through at least one transition. We call a node v such that $v \in P_v^{+}$ a *cyclic node*.

Now let us see the second proof.

Proof. Let $G = (V, B, I)$ be a finite graph in $DB_{\mathrm{f}\,Y}^{X}$. We can assume that $X = 1 (= \{\&\})$, recovering other cases by the graph constructors () and $\oplus$. Now we show the statement by induction on the natural number

$$|G| \overset{\text{def}}{=} |V| + 2 \times |V^{\text{cycle}}|$$

where V^{cycle} is the set of cyclic nodes.

If the root $I(\&)$ is cyclic, we define a graph

$$G' \overset{\text{def}}{=} (V \cup \{v_0\}, B', I) \in DB_{\mathrm{f}\,Y \cup \{\&y_0\}}$$

where v_0 and $\&y_0$ are fresh ones, and B' is defined as below.

$$\begin{aligned} B' \colon V \cup \{v_0\} &\to T\big(\mathcal{L}_{\epsilon} \times (V \cup \{v_0\}) + (Y \cup \{\&y_0\})\big) \\ v_0 &\mapsto \mathit{return}(\mathsf{Outm}\,(\&y_0)) \\ v &\mapsto T(g)(B(v)) \end{aligned}$$
$$\left(\begin{array}{l} g \colon \mathcal{L}_{\epsilon} \times V + Y \to \mathcal{L}_{\epsilon} \times (V \cup \{v_0\}) + (Y \cup \{\&y_0\}) \\ \mathsf{Edge}\,(l, I(\&)) \mapsto \mathsf{Edge}\,(l, v_0) \\ \text{the other case : embedding} \end{array} \right)$$

Then

$$G = \langle \&y_0 \rangle \mathbin{@} \mathbf{cycle}(\&y_0 := G'), \text{ and}$$
$$|G'| < |G|,$$

since, while G' has just one extra node, at least the root becomes non-cyclic.

When the root $I(\&)$ is not cyclic, consider $uf_{\varepsilon}(G)$; then as seen above, G is (strong bisimilar, hence) bisimilar to $uf_{\varepsilon}{}^{\text{-1}}(uf_{\varepsilon}(G))$, where recall that $uf_{\varepsilon}{}^{\text{-1}}$ is a composition of graph constructors. On the other hand, let us consider graphs G' occurring in

$$uf_{\varepsilon}(G) \in T(\mathcal{L}_{\epsilon} \times DB_{\mathrm{f}\,Y} + Y),$$

i.e., a graph G' in the minimum—then finite since T is finitary—set $P \subseteq DB_{\mathrm{f}\,Y}$ such that

$$uf_{\varepsilon}(G) \in T(\mathcal{L}_{\epsilon} \times P + Y).$$

Each G' in P is a subgraph of G, and the root of G' is a node in $P_{I(\&)}$ of G. Since $I(\&)$ is not cyclic, the accessible part of G' does not include $I(\&)$, hence $|G'| < |G|$. □