Structural Recursion for Querying Ordered Graphs

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

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

Abstract

Structural recursion, in the form of, for example, folds on lists and catamorphisms on algebraic data structures including trees, plays an important role in functional programming, by providing a systematic way for constructing and manipulating functional programs. It is, however, a challenge to define structural recursions for graph data structures, the most ubiquitous sort of 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. In this paper, we borrow from the database community the idea of structural recursion on how to restrict recursions on infinite unordered regular trees so that they preserve the finiteness property and become terminating, which are desirable properties for query languages. We propose a new graph transformation language called λ_{FG} for transforming and querying ordered graphs, based on the well-defined bisimulation relation on ordered graphs with special ε-edges. The language λ_{FG} is a higher order graph transformation language that extends the simply typed lambda calculus with graph constructors and more powerful structural recursions, which is extended for transformations on the sibling dimension. It not only gives a general framework for manipulating graphs and reasoning about them, but also provides a solution to the open problem of how to define a structural recursion on ordered graphs, with the help of the bisimilarity for ordered graphs with ε-edges.

Categories and Subject Descriptors CR-number [*subcategory*]: third-level; D.3.2 [*Programming Languages*]: Language Classifications—Specialized application languages; E.1 [*Data Structures*]: Graphs and networks

Keywords Structural Recursion, Ordered Graphs, Graph Query Language, Bisimulation, Optimization

1. Introduction

Structural recursion, in the form of, for example, folds on lists and catamorphisms (Meijer et al. 1991) on algebraic data structures including trees, plays an important role in functional programming, by 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 sort of data in computing. This is because, unlike lists and trees, graphs are essentially not inductive and cannot be formalized as an initial algebra (Gibbons 1995). Many attempts have been made to resolve this problem by using special trees to represent graphs so that structural recursion on trees can be used to manipulate graphs. This includes the work on representing graphs by trees with specific pointers (Hamana 2009; Dal Zilio et al. 2004; Oliveira and Cook 2012) and by trees with embedded functions (Fegaras and Sheard 1996). However, these attempts have not been so successful, because of the gap between trees (with specific pointers/embedded functions) and graphs, and they require programmers to bridge the gap by explicitly programming these specific pointers and embedded functions.

Would it be possible to define a structural recursion on graphs as if the graphs were trees (special graphs), while using it to manipulate general graphs? Yes, this has been proved to be possible to some extent in the database community, where UnCAL (and its user-level surface syntax UnQL) (Buneman et al. 2000) was introduced to provide a powerful querying method through structural recursion on finite graphs. The graph model of UnCAL is an unordered graph (whose outgoing edges are not ordered), which can be treated as (unordered) *regular trees* (Ginali 1979) given a suitable definition of bisimulation. The benefit of bisimulation is that structural recursion on regular trees can be used for graphs, and moreover, it leads to an interesting and important feature of structural recursion on graphs: *bulk semantics*. With bulk semantics, a structural recursion can be evaluated by first processing all edges of the input graph *in parallel* and then combining the results. Bulk semantics relies on the use of ε-edges (like ε transitions in the labeled transition system) in graphs and provides a smart way of treating shared nodes and cycles in graphs.

Despite usefulness of structural recursion in querying unordered graphs, there are two major limitations with UnCAL. One is that UnCAL can treat only unordered graphs; it cannot treat other graph models, such as the widely used ordered graphs in which the outgoing edges of a node are ordered, e.g., in EMF Ecore, MOF, and KM3 (Jouault and Bézivin 2006). Another limitation is the lack of expressive power of transformations in the sibling dimension. For example, when the edges of input graphs are labeled with natural numbers, we cannot write a transformation in UnCAL that extracts all edges labeled with the average number on the labels of all siblings.

At first sight, it seems that the first limitation could be easily overcome by encoding ordered graphs as unordered ones with suitable edge labels, say by using hd and tl to represent the first branch and the rest of the branches, respectively. However, there is a fatal problem with this hd-tl encoding. It is *unsound* in the context of UnCAL where ε-edges must be taken into account for the graph construction and structural recursion. Figure 1 shows

* Current affiliation is The University of Tokyo.

ICFP '13, September 25–27, 2013, Boston, MA, USA.

ACM 978-1-4503-2326-0/13/09.
http://dx.doi.org/10.1145/2500365.2500608

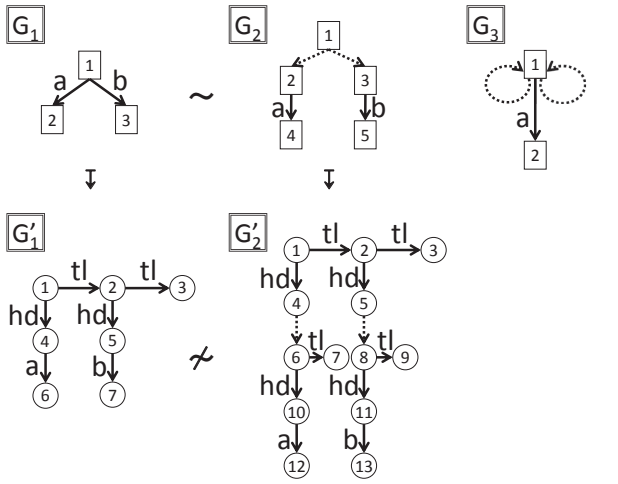


Figure 1. Examples of Head-tail Encoding and Its Unsoundness

an encoding example, where the numbers on the nodes have no particular meaning, since we consider bisimulation. 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 (denoted by $\sim$ in the figure), but the corresponding encoded graphs G'_1 and G'_2 ($\mapsto$ in the figure denotes the encoding) are not bisimilar at all.

One might think that we could do the encoding after deleting ε-edges. However, as the graph G_3 in Figure 1 shows, ε-edge elimination of some ordered graphs could produce *infinite width* graphs, and this infinite-width problem is not trivial to solve (see Section 3). Even if we could come up with an encoding, we would still have to show consistency and soundness of the encoding and the corresponding decoding and provide a suitable notion of bisimilarity for ordered graphs having ε-edges, which is very subtle as well (see Section 3). Moreover, we would need an effective way to guarantee that UnCAL transformations map well-encoded graphs to well-encoded graphs that can be successfully decoded later. All these suggest it would be better to design a new language that treats ordered graph directly.

In fact, it has been an *open problem* for more than ten years as to whether the definition of structural recursion in UnCAL can be modified so that it treats ordered graphs rather than unordered ones (the issue was first raised in the conclusion of the paper (Buneman et al. 2000)):

> *... there are some important and interesting areas of research that may well bear fruit. In connection with XML, 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 solution to this problem; i.e., we define a new structural recursion for ordered graphs. For this, we define the subtle notion of bisimilarity for ordered graphs having ε-edges, and prove *bisimulation genericity* (well-definedness with respect to bisimilarity) of our structural recursion. Note that our (and ref. (Buneman et al. 2000)'s) notion of bisimilarity is different from that of weak bisimilarity used in process algebra. Weak bisimilarity does not fit our purpose, because, for example, the graph union operator (works like the list append in the case of ordered lists) would not be associative.

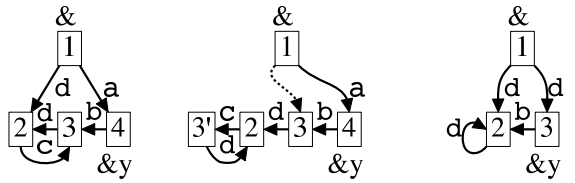
We propose λ_{FG}, a powerful higher-order graph transformation language, which is an extension of the simply typed lambda calculus with graph constructors and a more powerful structural recursion for manipulating ordered graphs than what is available in UnCAL. The main contributions of this paper can be summarized as follows.

- We propose a novel definition of bisimilarity between ordered graphs having ε-edges, forming the semantic foundation for λ_{FG}. Specifically, we show that the branch order is not necessarily finite but of *countable linear order*, even for finite graphs, and clarify that the combination of ε-edges and cycles induces such a countable linear order on the branchings, which would cause issues that do not occur in the unordered case. We show that we can decide whether an ordered graph is empty and whether we can eliminate all ε-edges for a finite ordered graph in such a way that the finite width property of the graph is kept.
- We show that graph constructors and structural recursion on unordered graphs can be adapted to those of ordered ones, forming the core syntax of λ_{FG}. Specifically, we define a more powerful structural recursion on ordered graphs, which can describe various graph queries including those on the *sibling dimension*. We also prove that (1) any graph query in λ_{FG} is *bisimulation generic* in the sense that it returns bisimilar results for bisimilar inputs, and (2) any graph query in λ_{FG} *terminates*, transforming *node-finite* graphs to *node-finite* graphs.

  Unlike UnCAL, we also define bisimilarity for higher order functions and prove bisimulation genericity of our structural recursion as a higher order function; this is the key to extending our language to a simply typed lambda calculus, and it makes it clear how we can extend it to rich type systems such as polymorphic/dependent type systems. Moreover, our proof of bisimulation genericity is clearer than the original proof given in (Buneman et al. 2000); this should make further extensions of structural recursion for graphs easier.
- We have implemented an interpreter for λ_{FG} that is available at `http://www.biglab.org/src/lambdaFG/`, and we have tested all the examples in this paper. In addition, we demonstrate (but have not implemented) that structural recursions are suitable for query reasoning and query optimization such as fusion and tupling transformations. This shows that our embedding of structural recursions into lambda calculus would be a good foundation to overcome the known impedance mismatch problem that is often raised from a gap between two languages when a query language is used in a general-purpose one for developing a practical application.

Note that in this paper, the "ordered" in "ordered graph" refers to the order on the branchings (rather than, say, the order on the nodes), and "graphs" generally refers to "ordered graphs" that can have ε-edges.

Organization of the Paper We shall start with an overview of λ_{FG}, a new graph transformation language for querying ordered graphs in Section 2. In Section 3, we discuss the first key technical contribution of this paper, defining ordered graphs, bisimilarity for ordered graphs having ε-edges, and ε-elimination. In Section 4, we give the semantics of our structural recursion and λ_{FG} and prove its bisimulation genericity. We demonstrate how to reason about graph transformations and discuss the expressive power of λ_{FG} in Section 5. Finally, we discuss related work in Section 6 and conclude the paper in Section 7.



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

Figure 2. Examples of Graphs

$$
\begin{array}{rcll}
\sigma & ::= & \sigma \to \sigma \mid \sigma + \sigma \mid \sigma \times \sigma & \{ \text{ function, coproduct, product types } \} \\
& | & \mathbf{List}(\sigma) \mid \mathbf{Bool} & \{ \text{ list and boolean types } \} \\
& | & \mathbf{Label} \mid \mathbf{G}^X_Y & \{ \text{ label and graph types } \} \\
e & ::= & x \mid \lambda x.e \mid e\ e \mid \mathbf{case}\ e\ \mathbf{of}\ \mathbf{in_l}(x) \to e\ \mathbf{or}\ \mathbf{in_r}(y) \to e & \\
& | & \mathbf{in_l}\ e \mid \mathbf{in_r}\ e \mid (e,e) \mid \mathbf{pr_l}e \mid \mathbf{pr_r}e & \{ \text{ terms of lambda calculus } \} \\
& | & \mathbf{nil} \mid \mathbf{cons}(e,e) \mid \mathbf{foldr}(e,e) \mid ... & \{ \text{ functions for lists } \} \\
& | & \mathbf{if}\ e\ \mathbf{then}\ e\ \mathbf{else}\ e & \{ \text{ conditional } \} \\
& | & a \mid e = e & \{ \text{ labels } (a \in \mathcal{L}) \text{ and label equality } \} \\
& | & \underline{[]} \mid e \mathbin{\underline{+\!\!+}} e \mid \underline{[e:e]} \mid \underline{[\&y]} \mid \&x := e \mid () \mid e \oplus e & \\
& | & e\ @\ e \mid \mathbf{cycle}(e) & \{ \text{ graph constructors } \} \\
& | & \mathbf{isEmpty}(e) & \{ \text{ graph emptiness checking } \} \\
& | & \mathbf{srec}(e,e) & \{ \text{ structural recursion functions } \}
\end{array}
$$

Figure 3. λ_{FG} Language

2. Overview of λ_{FG}

We start with an overview of our transformation language λ_{FG} for transforming (querying) ordered graphs. We introduce our graph model to which closed terms of graph types in λ_{FG} are interpreted. Then we present the syntax and typing rules of λ_{FG} and explain how to represent graphs and write transformations for graphs in λ_{FG}.

2.1 Ordered Graphs

Graphs treated in λ_{FG} are multi-rooted, directed, and edge-labeled. The graph model in λ_{FG} has three prominent features: *ε-edges*, *markers*, and *graph concatenation*. An ε-edge represents a shortcut between the two nodes and behaves like the ε-transition in automata. Nodes may be marked with *input* and *output markers*. These markers are used as an interface to other graphs through ε-edges. Graph concatenation sequentially aligns two graphs in a given order.

Formally, we define an ordered graph as follows. We write $\mathcal{L}$ for a set of *labels* and $\mathcal{L}_\varepsilon$ for $\mathcal{L} \cup \{\varepsilon\}$. Let X and Y be finite sets of input and output markers, respectively; we add the prefix & for markers like $\&x$. Accordingly, an *ordered graph* G (or just *graph*) is defined by a triple (V, B, I), where

- V is a set of *nodes*,
- $B\colon V \to \mathit{List}(\mathcal{L}_\varepsilon \times V + Y)$ is a *branch function* mapping a node to a list of *branches*: a branch in $\mathcal{L}_\varepsilon \times V + Y$ is either a *labeled edge* $\mathsf{Edge}(l, v)$ or an *output marker* $\mathsf{Outm}(\&y)$, and
- $I\colon X \to V$ is a function which determines the *input nodes* (also called *roots*) of the graph.

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

Example 1 (Ordered Graph)**.** The ordered graph in Figure 2(a), where the branches are ordered for each node, is represented as (V, B, I), where

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

In our graph model, every node in the range of function I is called a root. In Example 1, node 1 is the root node. Even though it is called a "root", it may have incoming edges. However, we can convert any graph having such edges incoming to the root into graphs bisimilar to the original one but having no such incoming edges.

We shall write $\mathcal{G}^X_Y$ for the set of ordered graphs with the input marker set X and the output marker set Y. We say that a graph is *finite* if V is a finite set and write $\mathcal{G}_{\mathrm{f}}{}^X_Y$ for the set of finite ordered graphs. When we say just "finite", it always means finiteness on the set of nodes. We allow a graph to have multiple roots, as a multi-rooted graph is to a forest what a single-rooted graph is to a tree. For single-rooted graphs, we often use the *default marker* & to indicate the root and use $\mathcal{G}_Y$ to denote $\mathcal{G}_Y^{\{\&\}}$.

2.1.1 Graph Equivalence

λ_{FG} uses bisimilarity (also called value-equivalence) as the graph equivalence for ordered graphs. For instance, the graphs G and G' in Figures 2(a) and 2(b) are equivalent. In G', nodes 3 and $3'$ are bisimilar because both nodes only have one outgoing edge labeled d to node 2. Also in G', there is an ε-edge (denoted by the dotted line) from node 1 to node 3, which can be eliminated while maintaining bisimilarity by replacing the ε-edge with an outgoing edge labeled d from node 1 to node 2. The parts that are unreachable from the roots are disregarded. The formal definition of bisimilarity for ordered graphs having ε-edges, which is one of the important results in this paper, is described in Section 3.

Note that the notion of equivalence based on bisimilarity influences the expressiveness of our query language. For example, since we do not distinguish bisimilar graphs like in Figures 2(a) and 2(b) —they have different numbers of nodes—, we cannot count the number of nodes of a graph.

2.2 Syntax of λ_{FG}

The syntax of λ_{FG} is given in Figure 3, and its typing rules are given in Figure 4, though we have omitted some standard typing rules. From here onwards, we will explain the syntax, but will omit the standard explanations for lambda terms and conditionals. We will explain graph constructors and structural recursion in detail in Sections 2.2.1 and 2.2.2 and give their formal semantics in Sections 3.4 and 4. We use the underlined syntax to distinguish certain graph constructors from ordinary list constructors.

$\mathbf{List}(\sigma)$ is the usual list type whose elements belong to σ; $\mathbf{nil}$, $\mathbf{cons}(e, e)$, and $\mathbf{foldr}(e, e)$ are standard list functions, and we can add any convenient list functions. These are used for specifying d in $\mathbf{srec}(e, d)$.

Note that we do not include the label ε in the syntax; ε-edges are only used in the semantics. Equality only applies to labels.

We have prepared a set of markers Marker; the metavariables X, Y, and so on denote finite subsets of Marker. Note that we assume type annotations for bound variables, graph constructors, and structural recursions: e.g., $\lambda x^\sigma.e$, $\underline{[]}_Y$, and $\mathbf{srec}_{X,Y,Z}(e, d)$; but we will omit them in this presentation for simplicity. We have a type inference procedure to omit marker set annotations in a way similar to our previous work for UnCAL (Hidaka et al. 2012).

The set of markers α in the type of d is an "abstract" set of markers, which can be viewed as being similar to a type variable in

$$\frac{(a \in \mathcal{L})}{\vdash a\colon \mathbf{Label}} \qquad \frac{\vdash e_1\colon \mathbf{Label} \quad \vdash e_2\colon \mathbf{Label}}{e_1 = e_2 : \mathbf{Bool}}$$

$$\frac{}{\vdash \underline{[]}\colon \mathbf{G}_Y} \qquad \frac{\vdash e_1\colon \mathbf{G}_Y^X \quad \vdash e_2\colon \mathbf{G}_Y^X}{\vdash e_1 \mathbin{\underline{+\!\!+}} e_2\colon \mathbf{G}_Y^X} \qquad \frac{\vdash e_1\colon \mathbf{Label} \quad \vdash e_2\colon \mathbf{G}_Y}{\vdash \underline{[}e_1\colon e_2\underline{]}\colon \mathbf{G}_Y}$$

$$\frac{(\&y \in Y)}{\vdash \underline{[}\&y\underline{]}\colon \mathbf{G}_Y} \qquad \frac{\vdash e\colon \mathbf{G}_Y}{\vdash \&x := e\colon \mathbf{G}_Y^{\{\&x\}}} \qquad \frac{}{\vdash ()\colon \mathbf{G}_Y^{\emptyset}}$$

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

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

$$\frac{\begin{array}{c}\vdash e\colon \mathbf{Label}\times\mathbf{G}_Y \to \mathbf{G}_Z^Z \\ \vdash d\colon \mathbf{List}(\mathbf{G}_{Z\times\alpha}^Z + \mathbf{G}_{Z\times Y}^Z) \to \mathbf{G}_{Z\times\alpha+Z\times Y}^Z\end{array}}{\vdash \mathbf{srec}(e,d)\colon \mathbf{G}_Y^X \to \mathbf{G}_{Z\times Y}^{Z\times X}}$$

(Note that we have omitted the typing environments Γ in each rule.)

Figure 4. Typing Rules for Graph-Related Expressions in λ_{FG}

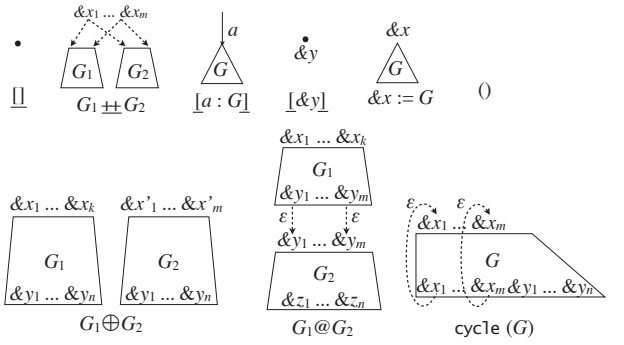


Figure 5. Graph Constructors

polymorphic lambda calculi. This polymorphism in the semantics of our structural recursion is explained in Section 4.

The graph type $\mathbf{G}_Y^X$ is interpreted as a set $\mathcal{G}_{\mathrm{f}Y}^X$ of *finite* graphs. Similarly to the notation given in Section 2.1 for sets of graphs, we use $\mathbf{G}_Y$ to denote $\mathbf{G}_Y^{\{\&\}}$. The predicate $\mathbf{isEmpty}(e)$ is true if the graph obtained by e has no non-ε-edges in the accessible part.

2.2.1 Graph Construction

λ_{FG} provides useful graph constructors to build arbitrary finite ordered graphs. Figure 5 summarizes the constructors; let us see how type discipline on the input and output markers works for each constructor. First, $\underline{[]}$ 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, $G_1 \mathbin{\underline{+\!\!+}} G_2$ *concatenates* them by adding two branching ε-edges from each new root to the corresponding old roots of G_1 and G_2. Next, $\underline{[}a : G\underline{]}$ extends G with a new fresh root node pointing to the old root of G with an a-labeled edge; the constructor $\underline{[}\&y\underline{]}$ constructs a graph with a single node marked with an output marker $\&y$ (in (Buneman et al. 2000), $\underline{[}a : G\underline{]}$ and $\underline{[}\&y\underline{]}$ are denoted as $\{a : G\}$ and $\&y$, respectively). *Marker renaming* $\&x := G$ associates an input marker $\&x$ with the root node of G (here, G should have only one default input marker $\&$). $()$ 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, i.e., the set of nodes of $G_1 \oplus G_2$ is the disjoint union of those of G_1 and G_2 and branching of nodes (B) is the same as the original. Then, $G_1 @ G_2$ *appends* two graphs by replacing the output markers of G_1 with an ε-edge pointing to the corresponding input nodes of G_2, and $\mathbf{cycle}(G)$ replaces the output marker of G with ε-edges pointing to the corresponding input nodes of G to form cycles.

Note that this set of constructors is powerful enough to describe any finite ordered graphs (although such a description may not be unique). More precisely, for any finite graph, there is a term using graph constructors whose interpretation is bisimilar to the given graph. This can be shown in a quite similar way to that in (Buneman et al. 2000). For instance, the ordered graph in Figure 2(a) can be constructed as follows:

$$\begin{array}{l}\&n_1 \mathbin{@} \mathbf{cycle}((\&n_1 := \underline{[}\mathtt{d} : \underline{[}\&n_2\underline{]}\underline{]} \mathbin{\underline{+\!\!+}} \underline{[}\mathtt{a} : \underline{[}\&n_4\underline{]}\underline{]}) \oplus \\ \qquad (\&n_2 := \underline{[}\mathtt{c} : \underline{[}\&n_3\underline{]}\underline{]}) \oplus \\ \qquad (\&n_3 := \underline{[}\mathtt{d} : \underline{[}\&n_2\underline{]}\underline{]}) \oplus \\ \qquad (\&n_4 := \underline{[}\mathtt{b} : \underline{[}\&n_3\underline{]}\underline{]} \mathbin{\underline{+\!\!+}} \underline{[}\&y\underline{]})).\end{array}$$

2.2.2 Structural Recursion

Structural recursion in λ_{FG} provides a powerful mechanism to describe transformations and queries over ordered graphs that guarantees the termination of the computation and preserves the finiteness of graphs.

In general, the typing rule of $\mathbf{srec}$ is the one given in Figure 4; but here, for ease of explanation, we will use the following simplified variant in which the input graph has no output marker.

$$\frac{\vdash e\colon \mathbf{Label}\times\mathbf{G}_\emptyset \to \mathbf{G}_Z^Z \quad \vdash d\colon \mathbf{List}(\mathbf{G}_{Z\times\alpha}^Z) \to \mathbf{G}_{Z\times\alpha}^Z}{\vdash \mathbf{srec}'(e,d)\colon \mathbf{G}_\emptyset^X \to \mathbf{G}_{Z\times\emptyset}^{Z\times X}}$$

This $\mathbf{srec}'$ is obtained as syntax sugar from the original $\mathbf{srec}$:

$$\mathbf{srec}'(e,d) \stackrel{\text{def}}{=} \mathbf{srec}(e, i \circ d \circ p)$$

where $p : \mathbf{List}(\mathbf{G}_{Z\times\alpha}^Z + \mathbf{G}_{Z\times\emptyset}^Z) \to \mathbf{List}(\mathbf{G}_{Z\times\alpha}^Z)$ discards elements in $\mathbf{G}_{Z\times\emptyset}^Z$ in an input list and $i : \mathbf{G}_{Z\times\alpha}^Z \to \mathbf{G}_{Z\times\alpha+Z\times\emptyset}^Z$ is the obvious isomorphism. Throughout this section, we will simply $\mathbf{srec}$ for $\mathbf{srec}'$.

Now let us briefly explain the semantics of $\mathbf{srec}(e,d)$; there are two different styles: *recursive semantics* and *bulk semantics*. Recursive semantics is more concise and useful for reasoning about not only the behavior of a function defined in a recursive way but also program transformation/optimization (This will be considered in Section 5.1). On the other hand, bulk semantics is useful for understanding how a graph is transformed on the whole, for parallel computation, and for proving the termination and finiteness-preserving properties. We will explain recursive semantics in a simple form here and formally describe these two semantics in Section 4.

We introduce a reasonable condition called *production-consumption compatibility (PCC)* on e and d so that we can give a clearer form of recursive semantics. The condition is explained in detail in Section 4.1, and the following examples in this section satisfy it. Furthermore, for simplicity, we consider the following case: $d = \mathbf{foldr}(\odot, \iota_\odot)$ for some monoid $(\odot, \iota_\odot)$ on $\mathbf{G}_{Z\times\alpha}^Z$; we consider general d in Section 4.1. The structural recursion function $f \stackrel{\text{def}}{=} \mathbf{srec}(e,d)$ satisfies the following equations (here, equality means the bisimilarity):

$$\begin{array}{lcl} f(\underline{[]}) & = & \iota_\odot \\ f(g_1 \mathbin{\underline{+\!\!+}} g_2) & = & f(g_1) \odot f(g_2) \\ f(\underline{[}l : g\underline{]}) & = & e(l,g) \mathbin{@} f(g) \\ f(\&x := g) & = & (\oplus_{\&z \in Z(\&z,\&x)} \&x := \underline{[}(\&z, \&)\underline{]}) \mathbin{@} f(g) \\ f(()) & = & () \\ f(g_1 \oplus g_2) & = & f(g_1) \oplus f(g_2). \end{array} \quad (1)$$

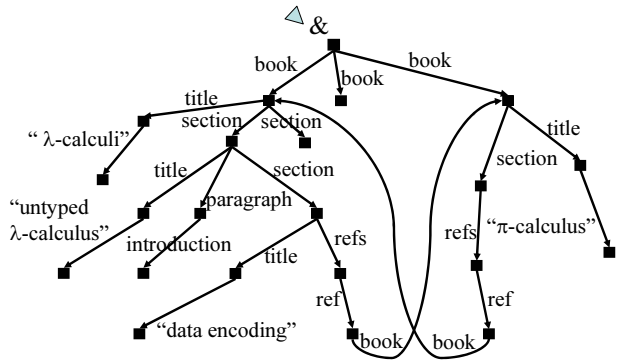


Figure 6. Ordered Graph Representation of Books

Though the above equations do not necessarily determine the functions f uniquely (up to bisimilarity), we can obtain a definition of f by the above equations: graphs can be regarded as infinite trees constructed by the above seven graph constructors (i.e., all the nine graph constructors but @ and **cycle**), and hence the above equations can be taken as a definition by a fixed-point operator.

Thus, f transforms graphs along the structures of graphs; e replaces a label with a new graph, and d (i.e., $\odot$) acts on branches and transforms in sibling directions. The fourth equation is just renaming of input markers and the fifth and sixth equations are cases of multi-rooted graphs; these three equations are straightforward, and so will be omitted in other forms of recursive semantics given later. In the case of the monoid $(\underline{+\!\!+}, \underline{[]})$—this satisfies PCC with any e—, the above recursive semantics is the same as given in UnCAL (Buneman et al. 2000) except that we treat ordered graphs rather than unordered graphs.

Though the above recursive semantics gives us a definition of the structural recursion, it is not obvious how to make the semantics terminating and that the semantics outputs finite (up to bisimilarity) graphs for given finite input graphs. It is the bulk semantics that shows the termination and finiteness of output graphs; and the bulk semantics satisfies the above equations.

In the following, we give several examples demonstrating the power of our language in transforming ordered graphs. Further examples are given in Sections 4.1 and 5.2.

Example 2. This example shows how we can manipulate edges of the graph and change its shape. The following structural recursion $a2d\_xc$ replaces all labels `a` with `d` and contracts `c`-labeled edges by using the function rc (*r*emove *c*).

$$\begin{aligned} a2d\_xc = \mathbf{srec}(rc, \mathbf{foldr}(\underline{+\!\!+}, \underline{[]})) \\ \quad \mathbf{where}\ rc(l, g) = \mathbf{if}\ l = \mathtt{a}\ \mathbf{then}\ \underline{[}\mathtt{d} : \underline{[}\&\underline{]}\underline{]} \\ \qquad\qquad \mathbf{else\ if}\ l = \mathtt{c}\ \mathbf{then}\ \underline{[}\&\underline{]}\ \mathbf{else}\ \underline{[}l : \underline{[}\&\underline{]}\underline{]} \end{aligned}$$

Applying the function $a2d\_xc$ to the graph in Fig. 2(a) yields the graph in Fig. 2(c). Note that $\underline{[}\&\underline{]}$ can be considered as a hole that will be later filled in with a recursive result. □

Example 3. This example demonstrates that our structural recursion can define transformations in the sibling direction, showing that it is more powerful than the structural recursion in UnCAL. The following function reverses the branches of each node of a graph.

$$\begin{aligned} revBranches = \mathbf{srec}(idE, \mathbf{foldr}(\hat{\underline{+\!\!+}}, \underline{[]})) \\ \quad \mathbf{where}\ idE(l, g) = \underline{[}l : \underline{[}\&\underline{]}\underline{]} \\ \qquad r_1 \hat{\underline{+\!\!+}} r_2 = r_2 \underline{+\!\!+} r_1 \end{aligned}$$

□

Example 4. This example demonstrates the usefulness of nested structural recursion. Figure 6 shows an ordered graph representation of a list of books that contains hierarchical structures with "section"-labeled edges. Since "section"s must be ordered and there would be reference links in books, we can regard books as ordered graphs. The following *nested* structural recursion toc, which is adapted from (Robertson et al. 2009), computes a table-of-contents of books in which sections can be arbitrarily nested:

$$\begin{aligned} toc = \mathbf{srec}(extractSection, \mathbf{foldr}(\underline{+\!\!+}, \underline{[]})) \\ \quad \mathbf{where}\ extractSection(l, g) = \\ \qquad \mathbf{if}\ l = \mathtt{section} \\ \qquad \mathbf{then}\ \underline{[}\mathtt{section} : (get\_title(g) \underline{+\!\!+} \underline{[}\&\underline{]})\underline{]} \\ \qquad \mathbf{else}\ \underline{[}\&\underline{]} \end{aligned}$$

where the function get_title, again defined by a structural recursion, results in the title of the section.

$$\begin{aligned} get\_title(g) = \mathbf{srec}(extractTitle, \mathbf{foldr}(\underline{+\!\!+}, \underline{[]})) \\ \quad \mathbf{where}\ extractTitle(l_1, g_1) = \\ \qquad \mathbf{if}\ l_1 = \mathtt{title} \\ \qquad \mathbf{then}\ \underline{[}\mathtt{title} : \mathbf{srec}(\lambda(l_2, g_2).\underline{[}l_2 : \underline{[]}, \mathbf{foldr}(\underline{+\!\!+}, \underline{[]}))\ (g_1)\underline{]} \\ \qquad \mathbf{else}\ \underline{[]} \end{aligned}$$

□

Example 5. We can define complex graph transformations by gluing together smaller ones. Suppose that we want to make a reverse version of the table-of-contents of a book. We can simply define it as

$$tocRev(g) = revBranches\ (toc(g)).$$

□

Before ending the review of λ_{FG}, it is worth remarking that compared with the structural recursion in UnCAL (Buneman et al. 2000), structural recursion in λ_{FG} can deal with ordered graphs and computations on the sibling dimension of graphs; in fact, **srec** in UnCAL does not have the function argument d. As will be seen later, the key to the success of our extension from unordered graphs to ordered ones is a new definition of bisimilarity for ordered graphs having ε-edges, under which (finite) ordered graphs are equivalent to (possibly infinite) regular trees up to bisimilarity.

In the rest of this paper, we will formally define our version of bisimilarity on ordered graphs, give a formal semantics for λ_{FG}, prove the three important properties of λ_{FG}, namely bisimulation genericity, termination, and finiteness-preservation, and demonstrate how to reason about graph transformations.

3. Ordered Graphs and Bisimilarity

As discussed in the introduction, direct manipulation of graphs is difficult. We will tackle this problem by regarding graphs as their equivalent trees so that we can manipulate graphs as if to manipulate trees. To this end, we need to define the semantic equivalence for the graph model of λ_{FG}: bisimilarity between ordered graphs. It should be noticed that ordered graphs have a big problem that does not occur with unordered graphs: i.e., ε-elimination might induce branches with an *infinite* width. In the following, we shall illustrate this problem and show how to define bisimilarity between ordered graphs. We will show an effective procedure to avoid infinite-width graphs.

3.1 Bisimilarity for Ordered Graphs

Let us show some examples of ordered graphs in order to get a feeling for the bisimilarity to be defined. Consider the graphs in Figure 7. Unfolding the graph G_s will yield an infinite tree (the graph in the middle), and performing "ε-elimination" on the infinite tree will give the graph on the right, whose branchings

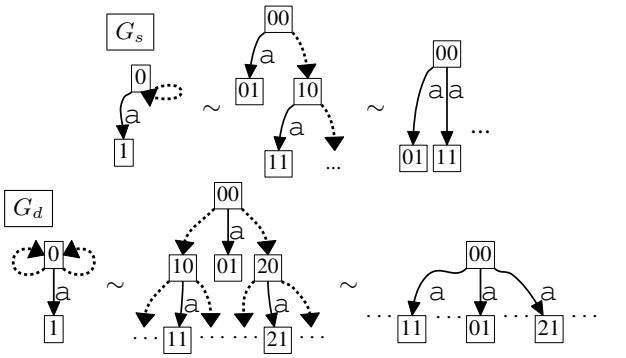


Figure 7. Graphs with Stream Branching and with Dense Branching

look like a stream. However, such infiniteness that occurs in ordered graphs having ε-edges is not just the stream type, as in the next example G_d. Unfolding G_d will yield the tree in the middle, and performing "ε-elimination" will give the graph on the right. This graph has a set of branches behaving 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.[1] It is worth noting that, if we were in an unordered setting where the branches are sets (that is, duplications of identical branches are ignored), G_s and G_d after ε-elimination would be bisimilar to the graph with only one branch; consequently, ε-elimination in this case would just delete the ε-loops.

Now let us define bisimilarity between ordered graphs. As can be seen from the above examples, the ε-elimination of an ordered graph might induce a countable width. Therefore, we shall first define a generalized notion of an "ordered graph with a countable width", and then proceed to define bisimilarity for such generalized graphs. To this end, for a set S, we extend the set of lists $List(S)(= \Sigma_{n\in\mathbb{N}} S^n)$ to that of the countable list $CList(S)$ defined as

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

where $\mathbb{N}$ is generalized to $\mathbb{L}$, the set of countable linear ordered sets up to order isomorphism [2]. Replacing $List$ with $CList$, we extend the notion of ordered graph to that of *ordered graph with a countable width*. The set of such extended graphs is denoted by $\mathcal{G}^{\mathrm{c}X}_Y$. Certainly, our original ordered graphs $\mathcal{G}^X_Y$ are a subset of $\mathcal{G}^{\mathrm{c}X}_Y$.

Once we become aware of the importance of countable linear order, our idea to define bisimilarity between graphs in $\mathcal{G}^{\mathrm{c}X}_Y$ is rather simple. Informally, given two graphs, a relation R is a bisimulation relation if for any two R-related nodes v and v', their corresponding *proper branch* sets are related through R. Here a proper branch of a node is defined as a path from v going through zero or more ε-edges and reaching a non-ε-edge.

Now we see the formal definitions. For a branching function $B(v)$, we use $|B(v)|$ to denote the countable linear ordered set L, call $i \in |B(v)|$ *a branch index* of a node v, and write $B(v).i \in \mathcal{L}_\varepsilon \times V + Y$ for the i-th branch.

Definition 6 (Proper Branch)**.** Let $G = (V, B, I) \in \mathcal{G}^{\mathrm{c}X}_Y$ and $v \in V$. The path starting from v

$$v\ (= v_0) \stackrel{\varepsilon}{\rightarrow}_{i_0} v_1 \ldots \stackrel{\varepsilon}{\rightarrow}_{i_{n-1}} v_n \rightarrow_{i_n}$$

is called a *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 $\mathrm{Pb}(G, v)$. □

Definition 7 (Order on Proper Branches)**.** Given two proper branches $p = (v \stackrel{\varepsilon}{\rightarrow}_{i_0} v_1 \ldots \stackrel{\varepsilon}{\rightarrow}_{i_{n-1}} v_n \rightarrow_{i_n})$ and $p' = (v \stackrel{\varepsilon}{\rightarrow}_{i'_0} v'_1 \ldots \stackrel{\varepsilon}{\rightarrow}_{i'_{n'-1}} v'_{n'} \rightarrow_{i'_{n'}})$, let their branch index sequences be $\tilde{p} \stackrel{\text{def}}{=} (i_0, \ldots, i_{n-1}, i_n)$ and $\tilde{p'} \stackrel{\text{def}}{=} (i'_0, \ldots, i'_{n'-1}, i'_{n'})$. We define $p \leq_{\mathrm{Pb}} p' \stackrel{\text{def}}{\Longleftrightarrow} \tilde{p} \leq_l \tilde{p'}$, where $\leq_l$ is the lexicographical order between branch index sequences. □

Now we are ready to define the bisimilarity.

Definition 8 (Bisimilarity)**.** For two graphs $G = (V, B, I)$ and $G' = (V', B', I')$ in $\mathcal{G}^{\mathrm{c}X}_Y$, a relation R between V and V' is called a *bisimulation relation*, if for any vRv', there is an order isomorphism $f\colon (\mathrm{Pb}(G, v), \leq_{\mathrm{Pb}}) \rightarrow (\mathrm{Pb}(G', v'), \leq_{\mathrm{Pb}})$ satisfying the following order-preserving property: For any proper branch $p = (v \stackrel{\varepsilon}{\rightarrow}_{i_0} \ldots v_n \rightarrow_{i_n}) \in \mathrm{Pb}(G, v)$ with $f(p) = (v' \stackrel{\varepsilon}{\rightarrow}_{i'_0} \ldots v'_{n'} \rightarrow_{i'_{n'}}) \in \mathrm{Pb}(G', v')$, we have

- Edge Correspondence: 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',
- Marker Correspondence: 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 $\&x \in X$, $I(\&x)\ R\ I'(\&x)$. □

3.2 Elimination of ε-Edges

The ε-edges are introduced (and necessary) in our internal semantics of graph transformation, but unnecessary ε-edges in the final result should be eliminated. The following definition leads to a procedure to eliminate ε-edges (for example by taking transitive closures) from the graphs and shows that the graph obtained by ε-elimination is bisimilar to the original graph.

Definition 9 (ε-elimination)**.** For a graph $G = (V, B, I) \in \mathcal{G}^{\mathrm{c}X}_Y$, the *$\varepsilon$-elimination* $\varepsilon\text{-elim}(G)$ of G is a graph $(V, B', I) \in \mathcal{G}^{\mathrm{c}X}_Y$ where $|B'(v)| \stackrel{\text{def}}{=} \mathrm{Pb}(G, v)$ and $B'(v).p \stackrel{\text{def}}{=} B(v_n).i_n$ for $p = (v \stackrel{\varepsilon}{\rightarrow}_{i_0} \ldots v_n \rightarrow_{i_n})$ in $|B'(v)|$. □

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

Proposition 10.

1. For any $G \in \mathcal{G}^{\mathrm{c}X}_Y$, $\varepsilon\text{-elim}(G)$ has no ε-edge, and G and $\varepsilon\text{-elim}(G)$ are bisimilar.
2. Two graphs $G, G' \in \mathcal{G}^{\mathrm{c}X}_Y$ are bisimilar if and only if $\varepsilon\text{-elim}(G)$ and $\varepsilon\text{-elim}(G')$ are bisimilar. □

3.3 Decidability of Empty and Finite-Width Graphs

In our graph transformation, i.e., any function from a graph type to a graph type in λ_{FG} like $tocRev$ in Example 5 the input graph is usually an ordered graph that has neither ε-edges nor infinite width, but the result may contain ε-edges in our context. In the following, we show that, given a graph with ε-edges, it is possible to decide whether there is a corresponding bisimilar ordered graph that has neither ε-edges nor infinite width, and there is a procedure to compute out such a bisimilar graph.

Let $\mathrm{FG}_{\not\varepsilon}$ be the set of finite ordered graphs (with finite width and possibly with ε-edges) that are bisimilar to some finite ordered graphs without ε-edge or infinite width. The following procedure

[1] In this paper, the term *countable* includes the finite case.

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

answers (decides) whether a finite ordered graph G is in FG$\not\varepsilon$ or not.

1. For each node accessible from a root, check if there is an ε-cycle—a cyclic path consisting only of ε-edges—on the node or not. If there is no ε-cycle, then G is in FG$\not\varepsilon$.
2. Otherwise, for every accessible node with an ε-cycle, if there is no proper branch, then graph G is in FG$\not\varepsilon$; otherwise, it is not in FG$\not\varepsilon$.

If G is in FG$\not\varepsilon$, we can effectively eliminate ε-edges; otherwise, it is impossible to eliminate ε-edges and keep the width finite. $\mathbf{isEmpty}(G)$ is true iff G has no non-ε edge in the accessible part, and thus, it is decidable.

Note that this procedure is useful for evaluating λ_{FG}; though we need ε-edges for the implementation—for structural recursion and for the efficiency of the graph calculation—, practical graphs in the real world have no ε-edges. If a user writes such a practical query, the result should be a graph in FG$\not\varepsilon$; if it is an incorrect query not intended by the user and then if the result has ε-edges, the above procedure can determine this and can warn the user. Note also that, in the FG$\not\varepsilon$ class, since we can eliminate ε-edges, we can use familiar effective procedures for checking bisimilarity and for obtaining the minimum graphs in a similar way to that of unordered graphs.

3.4 Graph Constructors

In Section 2.2.1, we have given an intuitive explanation for our graph constructors; the formal semantics of them should be obvious. As an example, we give the semantic definition of $G_1 \underline{+\!+} G_2$ (the formal semantics of all graph constructors can be found in Appendix A). For $G_i = (V_i, B_i, I_i) \in \mathcal{G}^X_Y$,

$$G_1 \underline{+\!+} G_2 \stackrel{\text{def}}{=} (V_1 \cup V_2 \cup \{v_1, \ldots, v_m\}, B', I')$$

where

$v_1, \ldots, v_m$ are fresh node identifiers

$\{\&x_1, \ldots, \&x_m\} = Dom(I_1)\ (= Dom(I_2))$

$B' = B_1 \cup B_2 \cup$

$\quad \{v_i \mapsto [\mathsf{Edge}(\varepsilon, I_1(\&x_i)), \mathsf{Edge}(\varepsilon, I_2(\&x_i))] \mid i{=}1, ..., m\}$

$I'(\&x_i) = v_i.$

Above, we assume V_1 and V_2 to be disjoint, which is realized by taking copies of them; recall that node identity is ignored in the context of bisimilarity.

One important property of the graph constructors is that they are bisimulation generic. As an example, consider $G_1 \underline{+\!+} G_2$; we want to show that $\underline{+\!+}$ is bisimulation generic in the sense that if $G_1 \sim G_1'$ and $G_2 \sim G_2'$, then $G_1 \underline{+\!+} G_2 \sim G_1' \underline{+\!+} G_2'$. We say that two graphs are *strongly bisimilar* if they are bisimilar without special treatment of ε-edges (i.e., ε-edges are considered as ordinary edges but labeled ε). It is quite straightforward to see that the graph concatenation constructor $\underline{+\!+}$ is strongly-bisimulation generic. Furthermore, we can see that $\varepsilon\text{-elim}(G_1 \underline{+\!+} G_2)$ is bisimilar to $\varepsilon\text{-elim}(\varepsilon\text{-elim}(G_1) \underline{+\!+} \varepsilon\text{-elim}(G_2))$. This means that strong-bisimulation genericity implies bisimulation genericity for this graph constructor. Similarly we can show other graph constructors are bisimulation generic. This leads to the following proposition.

Proposition 11 (Bisimulation Genericity of Graph Constructors)**.** All the graph constructors are strongly-bisimulation generic and also are bisimulation generic. □

4. Structural Recursion

Here, we give the semantics of our structural recursion; we modify structural recursion for unordered graphs UnCAL (Buneman et al. 2000) to deal with ordered graphs and extend it so that we can transform graphs also in the sibling direction. We start by giving *recursive semantics* of our structural recursion; this enables us to understand the behavior of functions by recursive reasoning. After that, we give the *bulk semantics* of the structural recursion to show the termination and finiteness preserving properties.

Next, we show that our structural recursion and every transformation in λ_{FG} are *bisimulation generic*; i.e., they returns bisimilar results for bisimilar inputs. This bisimulation genericity plays an important role in our framework, allowing us to reason about properties of graphs by using (possibly infinite) trees bisimilar to them. To this end, we extend the notion of bisimilarity to higher order functions in Section 4.3 and give the whole semantics of λ_{FG}.

After that, we will discuss implementation and termination property. Finally, we will summarize relations of conditions and results provided in this section.

Below, we often use the following *marker renaming graphs*: for a "marker renaming" function $f\colon X \to Y$, we define a graph $\lfloor f \rfloor \in \mathcal{G}^X_Y$ as $\lfloor f \rfloor \stackrel{\text{def}}{=} \bigoplus_{\&x \in X} \&x := \underline{[}f(\&x)\underline{]}$. As two use cases, for $G \in \mathcal{G}^X_Y$ with $f : X' \to X$, $\lfloor f \rfloor \mathbin{@} G \in \mathcal{G}^{X'}_Y$ is "input marker renaming" by f: an input marker $f(\&x)$ is replaced with $\&x$ and input markers of G that are not in the image of f are removed. Also, for $G \in \mathcal{G}^X_Y$ with $f : Y \to Y'$, $G \mathbin{@} \lfloor f \rfloor \in \mathcal{G}^X_{Y'}$ is "output marker renaming" by f: an output marker $\&y$ is replaced with $f(\&y)$.

4.1 Recursive Semantics

In Section 2.2.2, we described the recursive semantics of our structural recursion in the simple setting where we ignored output marker cases, assumed production-consumption compatibility (PCC), and restricted d to be the form of $\mathbf{foldr}(\odot, \iota_\odot)$. Here, we will give other forms of recursive semantics, relaxing these restrictions; but we still ignore output marker cases for the sake of simplifying the presentation. (For recursive semantics for the general d without ignoring output markers, see Appendix B.) Hence, in this subsection, the type of d is $\mathbf{List}(\mathbf{G}^Z_{Z\times\alpha}) \to \mathbf{G}^Z_{Z\times\alpha}$. (In bulk semantics, though, we will deal with also output marker cases.)

First, let us explain the polymorphism α in the type of d. This α plays two important roles: It is used to substitute a concrete set for α in order to express recursive semantics and define bulk semantics; it is also used to show bisimulation genericity of the bulk semantics by using relational parametricity. Formally, we should introduce α by formulating λ_{FG} as polymorphic typed lambda calculus as well as by introducing sets of markers as types, but to keep the presentation simple, we formulated λ_{FG} as a simply typed lambda calculus. In the following explanation of semantics, we will write d_X for the "type application" of d to X on α, and when we interpret $\mathbf{srec}(e, d)$, we introduce the universal quantifier $\forall\alpha.$ for d to bind α. We use the "type application" only in semantics for the above purposes, and do not use syntactically in λ_{FG} itself. Thus, λ_{FG} is essentially a rank-2 polymorphic lambda calculus; hence, type inference is decidable (Kfoury and Tiuryn 1990).

Now, let us consider recursive semantics of $f = \mathbf{srec}(e, d)$. For general e and d, the straightforward generalization of Equations (1) should be, just in form, as the following:

$$\begin{aligned} &f(\underline{[}l_1 : g_1\underline{]} \underline{+\!+} \cdots \underline{+\!+} \underline{[}l_n : g_n\underline{]}) \\ &= d_\emptyset\big([e(l_1, g_1) \mathbin{@} f(g_1), \ldots, e(l_n, g_n) \mathbin{@} f(g_n)]\big). \end{aligned} \tag{2}$$

This equation means that, when we start from the root of a graph, we have the branches $\underline{[}l_i : g_i\underline{]}$ under the root; then f first computes

$e(l_i, g_i)$ and recursively computes $f(g_i)$; they are connected by @ and finally are transformed in the sibling direction by using d.

However, for general e and d, this recursive semantics is not necessarily correct, i.e., the structural recursion functions defined by bulk semantics given later do not necessarily satisfy this equation. On the other hand, if we evaluate the above recursively, then, since the computation of d depends on $f(g_i)$, and since g_i may include a cyclic path returning to the original root, the evaluation may not terminate. Even if it terminates, the result is not necessarily bisimilar to the result by bulk semantics.

Instead, for general e and d, we have the following recursive semantics.

$$\begin{aligned} & f(\underline{[}l_1 : g_1\underline{]} \mathbin{\underline{+\!\!+}} \cdots \mathbin{\underline{+\!\!+}} \underline{[}l_n : g_n\underline{]}) \\ & = d_{\bar{n}}\big([e(l_1, g_1)@\lfloor \mathrm{in}_1 \rfloor, \ldots, e(l_n, g_n)@\lfloor \mathrm{in}_n \rfloor]\big) \\ & \quad @ \Big(\big(\lfloor \mathrm{iso}_1 \rfloor @ f(g_1)\big) \oplus \cdots \oplus \big(\lfloor \mathrm{iso}_n \rfloor @ f(g_n)\big)\Big) \end{aligned} \tag{3}$$

where $\bar{n} \stackrel{\text{def}}{=} \{1, ..., n\}$, $\mathrm{in}_i : Z \to Z \times \bar{n}$ is defined as $\mathrm{in}_i(\&z) \stackrel{\text{def}}{=} (\&z, i)$, and $\mathrm{iso}_i : Z \times \{i\} \cong Z$ is defined as $\mathrm{iso}_i((\&z, i)) \stackrel{\text{def}}{=} \&z$. This might look a bit complicated, but you can disregard marker-renaming parts $@\lfloor \mathrm{in}_i \rfloor$ and $\lfloor \mathrm{iso}_i \rfloor @$, which are not essential. A noteworthy point in Equation (5) is that d is applied before @ connects $e(l_i, g_i)$ to $f(g_i)$; in Equation (2), the order is reversed.

Now, the PCC condition is naturally introduced as the requirement that the recursive semantics (2) and (3) coincide, i.e., the order of application of d and connecting $e(l_i, g_i)$ with $f(g_i)$ by @ may be interchanged. Formally $e\colon \mathbf{Label} \times \mathbf{G}_\emptyset \to \mathbf{G}_Z^Z$ and $d\colon \mathbf{List}(\mathbf{G}_{Z \times \alpha}^Z) \to \mathbf{G}_{Z \times \alpha}^Z$ are called *production-consumption compatible (PCC)* if they satisfy the following equation:

$$\begin{aligned} & d_\emptyset\big([e(l_1, g_1) \mathbin{@} g'_1, \ldots, e(l_n, g_n) \mathbin{@} g'_n]\big) \\ & = d_{\bar{n}}\big([e(l_1, g_1) \mathbin{@} \lfloor \mathrm{in}_1 \rfloor, \ldots, e(l_n, g_n) \mathbin{@} \lfloor \mathrm{in}_n \rfloor]\big) \\ & \quad @ \Big(\big(\lfloor \mathrm{iso}_1 \rfloor @ g'_1\big) \oplus \cdots \oplus \big(\lfloor \mathrm{iso}_n \rfloor @ g'_n\big)\Big) \end{aligned}$$

Intuitively, the above condition says that d *consumes* only the information in the graphs *produced* by e and does not traverse beyond the output markers of the graphs produced by e. For example, if we write e such that any $e(l, G)$ has no output markers down to depth three, we can write d that can traverse graphs up to depth three. Note that, if d changes only the positions of graphs in an input list—such positions are regarded as positions of branches—and does not traverse each graph, then d satisfies the PCC condition with any e.

By definition, if e and d satisfy PCC, then $f = \mathbf{srec}(e, d)$ has the recursive semantics (2). Note that, in this paper, we call only the functions defined by the following equivalent semantics *structural recursion functions* (and denoted by $\mathbf{srec}(e, d)$): (i) the bulk semantics (Definition 13), (ii) the recursive semantics (3), and (iii) when PCC holds, the recursive semantics (2) and (1). For example, if e and d do not satisfy PCC, then in this paper we do not call functions defined by the recursive semantics (2) structural recursion functions even if they terminate.

Introducing d is our extension for sibling transformations from the original structural recursion in UnCAL. Note that, no matter whether we use the general recursive semantics (3) or the recursive semantics (2) with PCC, access by d is confined to the graphs produced by e anyway. This is the key restriction on our structural recursion for termination.

Example 12. In this example, d satisfies PCC with any e. To remove the even branches of a graph, we can apply $even\_remove \stackrel{\text{def}}{=} \mathbf{srec}(e, d)$ where $e(l, g) \stackrel{\text{def}}{=} \underline{[}l : \&\underline{]}$; and $d(\underline{[]}) \stackrel{\text{def}}{=} \underline{[]}$, $d(\underline{[}g\underline{]}) \stackrel{\text{def}}{=} g$, and $d(g :: g' :: gs) \stackrel{\text{def}}{=} g \mathbin{\underline{+\!\!+}} d(gs)$.

In the next example, e and d do not satisfy PCC, but we still can reason that this works as expected from the recursive semantics (3). To contract the even edges of a graph, from (3), it is enough that e is an "identity" and d contracts even labeled edges; i.e., we can apply $even\_contract \stackrel{\text{def}}{=} \mathbf{srec}(e, d)$ where

$$\begin{aligned} & e(l, g) \stackrel{\text{def}}{=} \underline{[}l : \&\underline{]}, \qquad d(\underline{[]}) \stackrel{\text{def}}{=} \underline{[]}, \qquad d(\underline{[}g\underline{]}) \stackrel{\text{def}}{=} g, \\ & d(g :: g' :: gs) \stackrel{\text{def}}{=} g \mathbin{\underline{+\!\!+}} p(g') \mathbin{\underline{+\!\!+}} d(gs), \\ & p \stackrel{\text{def}}{=} srec(\lambda(l, g).\underline{[}\&\underline{]}, \mathbf{foldr}(\underline{+\!\!+}, \underline{[]})) \ : \ \mathbf{G}^{\{\&\}}_{\{\&\} \times \alpha} \to \mathbf{G}^{\{\&\}}_{\{\&\} \times \alpha}. \end{aligned}$$

4.2 Bulk Semantics

Here, we give bulk semantics of the structural recursion $\mathbf{srec}(e, d)$ for a general d not assuming PCC nor ignoring output markers.

Before giving the formal definition of the bulk semantics of $\mathbf{srec}(e, d)$, we illustrate the behavior of the semantics in an example $a2d\_xc \stackrel{\text{def}}{=} \mathbf{srec}(rc, \mathbf{foldr}(\underline{+\!\!+}, \underline{[]}))$ (see Example 2 for rc). Mainly, an evaluation with the bulk semantics consists of three steps; we start with the input graph (a) in Figure 8.

1. *Applying the Map Computation on Edges with e*

   We apply the function rc (which renames a-labeled edges as d-labeled ones and contracts the c-labeled edges) to every l-labeled edge and the graph g following the edge, to yield a graph in $\mathcal{G}_Z^Z$. Let us call the graphs produced by an expression e *e-graphs*. Graph (b) is a graph whose edges are labeled by rc-graphs.

2. *Applying Map Computation on Nodes with d*

   For each edge—from a node v to a node v'—of Graph (b), we replace every output marker $\&z$ of the e-graph with a pair $(\&z, v')$, which will be used in the next "grouping" step. Next, for every node v of Graph (b), we use binary operator $\underline{+\!\!+}$ to combine all e-graphs which are labels of the branches of v. Let us call the graphs produced using d $(= \mathbf{foldr}(\underline{+\!\!+}, \underline{[]}))$ *d-graphs*; Graph (c) shows the result. Nodes 1 and 4 have more than one branch which need to be merged using $\underline{+\!\!+}$. After producing the d-graph for v, we replace each input marker $\&z$ with a pair $(\&z, v)$.

3. *Grouping New Graphs with ε-Edges*

   Now, for each node of Graph (b)—equivalently, for each node of the original graph—, we have a d-graph produced as above. We group all these d-graphs in Graph (c) by connecting every output marker $(\&z, v)$ of every d-graph to the corresponding input node of some d-graph by using an ε-edge. The root of the new graph is that of the d-graph on the original root node. Thereby, we get Graph (d) as an evaluation result of the structural recursion. Graph (e) is the graph obtained by performing ε-elimination on Graph (d). We remark that ε-elimination may increase the number of non-ε edges as this example. (We can further minimize Graph (e) if necessary, resulting the graph in Figure 2(c).)

Before applying a structural recursion function $\mathbf{srec}(e, d)$ to G, we have to perform ε-elimination on G; if $\varepsilon\text{-elim}(G)$ has infinite width, we raise an error. Without the ε-elimination, if we extend the bulk semantics to treat input graphs having ε-edges by extending e to $\bar{e}(\varepsilon, g) \stackrel{\text{def}}{=} \lfloor \mathrm{id}_Z \rfloor$, then the $\mathbf{srec}(\bar{e}, d)$ is not necessarily bisimulation generic, as shown by the following counter example. The two graphs on the left are bisimilar, but if we apply the structural recursion function $even\_remove$ in Example 12, the resulting graphs are not bisimilar.

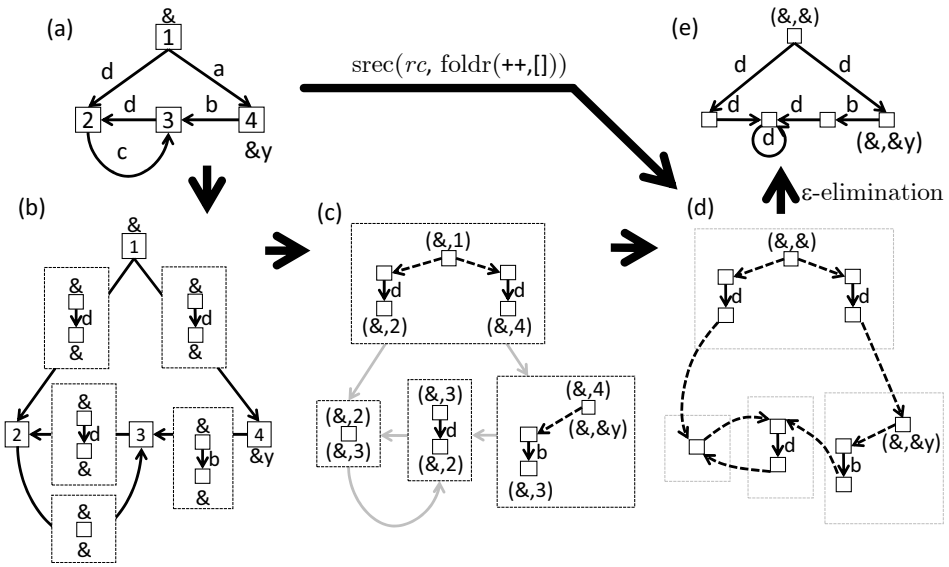
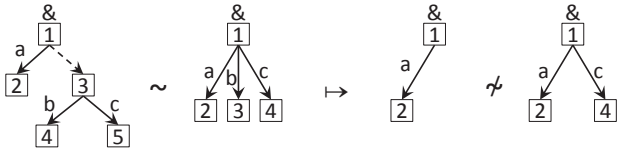


Figure 8. Bulk Semantics of Structural Recursion: An Example of $\mathbf{srec}(rc, \mathbf{foldr}(\underline{+\!\!+}, \underline{[]}))$



Though, if $d = \mathbf{foldr}(\underline{+\!\!+}, \underline{[]})$ or $d = \mathbf{foldr}(\hat{+\!\!+}, \underline{[]})$ (reverse function), then we can show the bisimulation genericity without performing ε-elimination.

Now let us give the formal definition of the bulk semantics of the structural recursion. Note that the following definition is more concise than the one given in (Asada et al. 2012), which mimics the original bulk semantics of UnCAL given in (Buneman et al. 2000); and this simplification makes the proof of bisimulation genericity clearer.

Definition 13 (Bulk Semantics)**.** For functions

$$e\colon \mathcal{L}\times\mathcal{G}_Y \to \mathcal{G}_Z^Z \quad d\colon \forall\alpha.\mathit{List}(\mathcal{G}_{Z\times\alpha}^Z + \mathcal{G}_{Z\times Y}^Z) \to \mathcal{G}_{Z\times\alpha+Z\times Y}^Z,$$

a *structural recursion function* $\mathbf{srec}(e,d)\colon \mathcal{G}_Y^X \to \mathcal{G}_{Z\times Y}^{Z\times X}$ is defined as follows.

Let $G = (V, B, I)$ be a graph in $\mathcal{G}_Y^X$. As explained above, we perform ε-elimination on G in order to ensure bisimulation genericity; hence, we assume G has no ε-edges. Let us define $\tilde{e}\colon \mathcal{L}\times V \to \mathcal{G}_Z^Z$ as $e \circ (\mathrm{id}_{\mathcal{L}} \times G|_{(\cdot)})$, where $G|_{(\cdot)}\colon V \to \mathcal{G}_Y$ extracts the subgraph of G whose root is a given node: i.e., $G|_v \stackrel{\text{def}}{=} (V, B, \{\&\mapsto v\})$.

Then, as in the first step explained above, we construct a new branching function $B' \stackrel{\text{def}}{=} \mathbf{map}(\langle\tilde{e}, \pi_2\rangle + \mathrm{id}_Y) \circ B$ of type

$$V \xrightarrow{B} \mathit{List}(\mathcal{L}\times V + Y) \xrightarrow{\mathbf{map}(\langle\tilde{e},\pi_2\rangle+\mathrm{id}_Y)} \mathit{List}(\mathcal{G}_Z^Z \times V + Y).$$

This corresponds to a step producing Graph (b) in Figure 8.

As for the second step, first, we construct a function $d'_\alpha \stackrel{\text{def}}{=} d_\alpha \circ \mathbf{map}(p+q)$ of type

$$\mathit{List}(\mathcal{G}_Z^Z \times \alpha + Y) \xrightarrow{\mathbf{map}(p+q)} \mathit{List}(\mathcal{G}_{Z\times\alpha}^Z + \mathcal{G}_{Z\times Y}^Z) \xrightarrow{d_\alpha} \mathcal{G}_{Z\times\alpha+Z\times Y}^Z$$

where $p(G, \&a) \stackrel{\text{def}}{=} G \mathbin{@} \lfloor \&z \mapsto (\&z, \&a) \rfloor$ and $q(\&y) \stackrel{\text{def}}{=} \lfloor \&z \mapsto (\&z, \&y) \rfloor$. Next, by using the following bijective correspondence,

$$\mathit{oplus} : (V \to \mathcal{G}_{Z\times V+Z\times Y}^Z) \xrightarrow{\cong} \mathcal{G}_{Z\times V+Z\times Y}^{Z\times V},$$
$$\mathit{oplus}(f) \stackrel{\text{def}}{=} \bigoplus_{(\&z,v)\in Z\times V} (\&z, v) := (\underline{[\&z]} \mathbin{@} f(v))$$

we obtain a graph $G' \stackrel{\text{def}}{=} \mathit{oplus}(d'_V \circ B') \in \mathcal{G}_{Z\times V+Z\times Y}^{Z\times V}$, which corresponds to (c) in Figure 8; then, in correspondence with (d) in Figure 8, we define

$$\mathbf{srec}(e,d)(G) \stackrel{\text{def}}{=} \lfloor \mathrm{id}_Z \times I \rfloor @ \mathbf{cycle}(G') \quad (\in \mathcal{G}_{Z\times X}^{Z\times X}).$$

□

It is worth remarking that the above bulk semantics is essentially *root-independent*: the whole part except for the final step—i.e., till constructing $\mathbf{cycle}(G')$—does not use the input function I.

The following is clear from the above bulk semantics.

Proposition 14. *If the functions* e *and* d *map finite graphs to finite graphs, so does* $\mathbf{srec}(e,d)$.

From the above bulk semantics, we can show obviously the termination property of λ_{FG}. We can implement the above bulk semantics in an obvious way: We may represent graphs (X, Y, V, B, I) using "set" and "map" (implemented as, e.g., balanced trees); X,Y, and V are as "set"s, and B and I are as "map"s. Then implementation of the bulk semantics is straightforward. It is clear that the computation of the bulk semantics terminates by the above definition and the finiteness on nodes and width of graphs, which is ensured by the above proposition. Our actual implementation in OCaml can be found at `http://www.biglab.org/src/lambdaFG/`; note that d in **srec** in the implementation is currently restricted to the form $\mathbf{foldr}(\odot, \iota_\odot)$.

4.3 Semantics of λ_{FG} and Bisimilarity on Functions

Before proving the bisimulation genericity of the structural recursion, we will summarize the whole semantics of λ_{FG}.

A requirement of our semantics is that it should guarantee that any transformation—including the one using higher order functions—in λ_{FG} must be bisimulation generic. To this end, we extend the notion of bisimilarity on graphs to bisimilarity

on functions in λ_{FG}. We have defined bisimilarity for graph types $\mathbf{G}_Y^X$, and we can use the equality relations for the other basic types. It is well known that, if we want to lift equivalence relations to function types, we need to switch from the notion of an equivalence relation to that of a *partial equivalence relation*, i.e., equivalence relation on some *subset* of the original set. This is because, in the current case, it is not true that all functions on $\mathcal{G}_{\mathrm{f}Y}^X$ are bisimulation generic, so we have to focus on its *subset* consisting of bisimulation generic functions.

Formally, for types σ of λ_{FG}, we define binary *logical relations* $\sim_\sigma$ from the above equivalence relations on the base types. Let us recall the logical relation $\sim_\sigma$ only in the essential case, i.e., function types: $\sigma = \sigma_1 \to \sigma_2$ (for other cases, one can consult (Mitchell 1996)). We define a binary relation $\sim_\sigma$ on $[\![\sigma]\!] \stackrel{\text{def}}{=} [\![\sigma_1]\!] \to [\![\sigma_2]\!]$ as

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

For any type σ, $\sim_\sigma$ becomes a partial equivalence relation on $[\![\sigma]\!]$, i.e., an equivalence relation on the subset $|\sim_\sigma| \stackrel{\text{def}}{=} \{x \in [\![\sigma]\!] \mid x \sim_\sigma x\}$. We say a function $f\colon [\![\sigma_1]\!] \to [\![\sigma_2]\!]$ *bisimulation generic* if f is in $|\sim_{\sigma_1 \to \sigma_2}|$.

Now from the Basic Lemma of the logical relation (Mitchell 1996), interpretations of all the terms are bisimulation generic if interpretations of all the constants (including constant functions) are bisimulation generic; accordingly, we can form a model (a cartesian closed category) of λ_{FG} such that the interpretations of types are the quotient sets of the partial equivalence relations. We have already shown the bisimulation genericity of graph constructors, and now we will show the same for the structural recursion.

4.4 Bisimulation Genericity of the Structural Recursion

We prove a stronger statement of bisimulation genericity than the original one in (Buneman et al. 2000), which proves only first-order bisimulation genericity, i.e., only bisimulation genericity on graph arguments. In addition, our proof is much clearer than the original naive proof. This simplification should make further extensions of structural recursion easier.

Theorem 15 (Bisimulation Genericity of $\mathbf{srec}$)**.** *The higher order function* $\mathbf{srec}$ *is bisimulation generic; i.e., for* $e_1 \sim e_2$*,* $d_1 \sim d_2$*, and* $G_1 \sim G_2$*,* $\mathbf{srec}(e_1, d_1)(G_1) \sim \mathbf{srec}(e_2, d_2)(G_2)$.

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

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

For $S : X_1 \nrightarrow X_2$ and $T : Y_1 \nrightarrow Y_2$, let us define the relation $\sim_T^S\colon \mathcal{G}_{Y_1}^{X_1} \nrightarrow \mathcal{G}_{Y_2}^{X_2}$: for G_i, let $(V_i, B_i, I_i) = \varepsilon\text{-elim}(G_i)$, then $G_1 \sim_T^S G_2$ if there exists a relation $R : V_1 \nrightarrow V_2$ such that the right diagram above commutes (i.e., each of the two squares is true). We write simply X for the diagonal relation between X and X, then note that $\sim_Y^X\colon \mathcal{G}_Y^X \nrightarrow \mathcal{G}_Y^X$ is the same as $\sim$.

Proof. Let G_i be (V_i, B_i, I_i). From $G_1 \sim G_2$, there exists $R : V_1 \nrightarrow V_2$ such that the following diagram commutes.

$$\begin{array}{ccccc} X & \xrightarrow{I_1} & V_1 & \xrightarrow{B_1} & List(\mathcal{L}\times V_1 + Y) \\ \| & & {\scriptstyle R}\downarrow & & \downarrow{\scriptstyle +List(\mathcal{L}\times R+Y)} \\ X & \xrightarrow[I_2]{} & V_2 & \xrightarrow[B_2]{} & List(\mathcal{L}\times V_2 + Y) \end{array} \tag{4}$$

Next, from the assumption that $e_1 \sim e_2$ and $d_1 \sim d_2$, the following diagram commutes.

$$\begin{array}{ccccccc} V_1 & \xrightarrow{B_1} & List(\mathcal{L}\times V_1 + Y) & \xrightarrow{\mathbf{map}(\langle e_1', \pi_2\rangle + \mathrm{id}_Y)} & List(\mathcal{G}_Z^Z \times V_1 + Y) & \xrightarrow{d_{1V_1}'} & \mathcal{G}_{Z\times V_1 + Z\times Y}^Z \\ {\scriptstyle R}\downarrow & & \downarrow{\scriptstyle List(\mathcal{L}\times R+Y)} & & \downarrow{\scriptstyle List(\sim_Z^Z \times R+Y)} & & \downarrow{\scriptstyle \sim_{Z\times R+Z\times Y}^Z} \\ V_2 & \xrightarrow[B_2]{} & List(\mathcal{L}\times V_2 + Y) & \xrightarrow[\mathbf{map}(\langle e_2', \pi_2\rangle + \mathrm{id}_Y)]{} & List(\mathcal{G}_Z^Z \times V_2 + Y) & \xrightarrow[d_{2V_2}']{} & \mathcal{G}_{Z\times V_2 + Z\times Y}^Z \end{array}$$

Note that to show the right square above, we also use the relational parametricity of d_i and hence of d_i'; see Definition 13 for the definition of d_i'. Then, for the above compositions, transposing V_i by using *oplus* as in Definition 13 yields

$$\begin{array}{ccc} 1 & \xrightarrow{G_1'} & \mathcal{G}_{Z\times V_1 + Z\times Y}^{Z\times V_1} \\ \| & & \downarrow{\scriptstyle +\sim_{Z\times R+Z\times Y}^{Z\times R}} \\ 1 & \xrightarrow[G_2']{} & \mathcal{G}_{Z\times V_2 + Z\times Y}^{Z\times V_2} \end{array} \quad \text{i.e.,} \quad G_1' \sim_{Z\times R+Z\times Y}^{Z\times R} G_2'. \tag{5}$$

After that, we can show that

$$\mathbf{cycle}(G_1') \sim_{Z\times Y}^{Z\times R} \mathbf{cycle}(G_2'), \tag{6}$$

and from the left square of the diagram (4) we get

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

i.e., $\mathbf{srec}(e_1, d_1)(G_1) \sim \mathbf{srec}(e_2, d_2)(G_2)$. □

For readers familiar with category theory, we will elaborate a bit on the general technique behind Equation (6). From the correspondence $Y \mapsto \mathcal{G}_Y$, we can construct a monad on $\mathbf{Set}$; then $\mathbf{cycle}$ becomes an *iteration operator* (dual of fixed point operator) in its Kleisli category. Furthermore, the iteration operator is uniform on values (i.e., functions). The above Equation (6) from Equation (5) is then the uniformity principle on relations (Hasegawa 2002).

4.5 Remark

We conclude this section with a summary of some conditions that we assumed occasionally. (Ignoring output marker cases is just for presentation and not essential.)

We use PCC to obtain more familiar form of recursive semantics (2), and moreover we assumed d to be in the fold form of monoid to obtain simpler recursive semantics (1). The PCC condition and its recursive semantics are also used for optimization given in Section 5.1.

The polymorphism of α of d is used to express recursive semantics and define bulk semantics, and its parametricity is used to prove the bisimulation genericity of the bulk semantics. The preprocessing of ε-elimination for input graphs of structural recursion functions is also used to prove the bisimulation genericity.

Termination and node-finiteness preserving properties of the structural recursion was shown via the bulk semantics, and essentially comes from the suitable restriction of recursion pattern; our structural recursion still has rich expressive power (see Section 5.2) and enables systematic optimization by the recursive semantics (see Section 5.1).

5. Discussion

5.1 Optimization of λ_{FG} Programs

Embedding graph queries into lambda calculus as structural recursions would make query optimization easier and more systematic. This is not only because existing optimization techniques for functional programs can be directly brought in, but also because structural recursion itself has nice algebraic properties for optimization.

Here, we illustrate the usefulness of structural recursion in reasoning and manipulating graph queries with two known transformation techniques, fusion and tupling.

Fusion Transformation Fusion (or called deforestation) (Meijer et al. 1991; Gill et al. 1993; Takano and Meijer 1995) is an important program transformation to turn a composition of two structural recursions into one so that unnecessary intermediate data passed from one structural recursion to another can be removed. The following is our fusion rule showing that a composition of a function f and a structural recursion $\mathbf{srec}(e, d)$ can be merged into one as long as the PCC condition holds for e and d:

$$f \circ \mathbf{srec}(e, d) \;=\; \mathbf{srec}(e' \circ e, d')$$

holds, provided that f is promoted on d and demoted on @; i.e., for some f', we have

$$f \circ d = d' \circ \mathbf{map}\ f' \quad \text{and} \quad f'(g \mathbin{@} r) = e'(g) \mathbin{@} f(r).$$

Tupling Transformation Tupling (Hu et al. 1997) is another important optimization technique. It eliminates multiple traversals of the same data. Our following tupling transformation rule reduces two traversals of G given by two structural recursions into one.

$$\begin{array}{l}(\&x_1 := \mathbf{srec}(e_1, d_1)(G)) \oplus (\&x_2 := \mathbf{srec}(e_2, d_2)(G)) \\ \quad = \ \mathbf{srec}(e, d)(G) \\ \quad \mathbf{where} \\ \qquad e\ (l, g) \ = \ (\&x_1 := (e_1(l, g) \mathbin{@} \underline{[}\&x_1\underline{]})) \\ \qquad\qquad \oplus \ (\&x_2 := (e_2(l, g) \mathbin{@} \underline{[}\&x_2\underline{]})) \\ \qquad d\ gs \ = \ (\&x_1 := d_1(\mathbf{map}\ (\lambda g.\underline{[}\&x_1\underline{]} \mathbin{@} g)\ gs)) \\ \qquad\qquad \oplus \ (\&x_2 := d_2(\mathbf{map}\ (\lambda g.\underline{[}\&x_2\underline{]} \mathbin{@} g)\ gs))\end{array}$$

The new structural recursion basically computes in parallel two graphs marked by $\&x_1$ and $\&x_2$. Note that the tuple construction and projection that are used in the ordinary tupling technique are encoded here by the disjoint union ($\oplus$) and marker renaming: a pair "(g_1, g_2)" is by $(\&x_1 := (g_1 \mathbin{@} \underline{[}\&x_1\underline{]})) \oplus (\&x_2 := (g_2 \mathbin{@} \underline{[}\&x_2\underline{]}))$, and a projection "$\pi_i(g)$" is by $\underline{[}\&x_i\underline{]} \mathbin{@} g$.

5.2 Expressive Power of λ_{FG}

In the previous sections, we described the definition and properties of λ_{FG}, and showed some interesting query examples. In this section, we discuss the expressive power of λ_{FG} from the user's point of view, to show that λ_{FG} can serve as a basis of query languages on ordered graphs under bisimulation.

As has already been demonstrated by Buneman et al. (2000) in the unordered setting, nesting of **srec** can produce a sequence of combinations of label and graph variable bindings $(\lambda(l_1, g_1), \lambda(l_2, g_2), \ldots)$. We can combine this nesting with conditionals on these variables to express variety of join conditions. For example, two nested structural recursions can extract subgraphs following consecutive edges with identical labels. Moreover, regular path expressions of labels can be expressed by **srec** with a body e generating multiple roots via tupling (Hu et al. 1997), as shown in Section 5.1. Such an e can be systematically constructed from the regular path expressions of labels via finite state automata, as in Buneman et al. (2000) (while NFA is used in (Buneman et al. 2000), we need DFA to make the result well-defined). These kinds of queries in λ_{FG} can be concisely expressed by using a more user-friendly syntactic sugar like UnQL for UnCAL in (Buneman et al. 2000) (although we do not give any concrete desugaring rules here) so that users do not have to write nested structural recursions with multiple root expressions manually. For example,

$$\mathbf{select}\ e\ \mathbf{where}\ \underline{[}\mathit{path}{:}g\underline{]}\ \mathbf{in}\ \mathit{db}$$

binds the graph variable g to every subgraph pointed to by a regular path expression path from the root of the graph that db is bound to, and will join ($\underline{+\!\!+}$) the graphs generated by the expression e that refers the graph variable g.

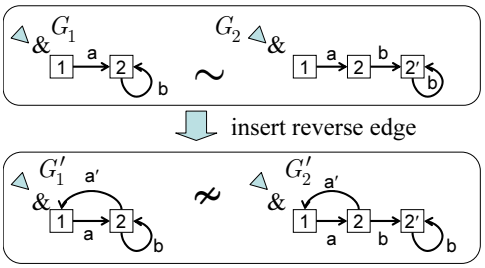
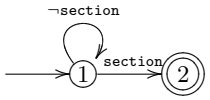


Figure 9. insertion of reverse edge

It is worth noting that, in case of extracting subgraphs under regular path pattern, we just shortcut the edges with unmatching labels like in the **else** clause of $\mathit{extractSection}$ in Example 4. If there is a cycle of these unmatching labels (labels other than `section` in the example) above the matching subgraph, then the cycle turns into an ε-cycle, meaning that the result of query represents infinite number of that subgraphs. This infinity is no surprising since original input graph essentially includes infinite number of that subgraphs when the cycle is unfolded. The infinity in the result is reported by ε-elimination error in λ_{FG}. Interestingly, this exhibits the expressive power of λ_{FG} to *detect cyclic patterns* specified by the automata mentioned above. For example, $\mathit{extractSection}$ corresponds to automata on the right, which includes a cycle. If this cycle and the ones in the input graph matches, then an ε-cycle is produced.



Expressive Power on Sibling Dimension With the parameter d of type $\mathbf{List}(\mathbf{G}^Z_{Z\times\alpha}+\mathbf{G}^Z_{Z\times Y}) \to \mathbf{G}^Z_{Z\times\alpha+Z\times Y}$ in **srec** we can express not only queries such as extracting subgraphs as mentioned above, but also those manipulating siblings of nodes, like reversal (Example 2), removing and contracting even-numbered siblings (Example 12).

Expressiveness under Bisimilarity Equivalence Since our language is bisimulation generic, targeting graphs under bisimilarity equivalence, queries that distinguish bisimilar graphs are not expressible. For example, we cannot count the number of nodes, since two bisimilar graphs may have different numbers of nodes. As another example, we cannot insert a 'reverse' edge. For example, in Figure 9, the graph G_2 is obtained from G_1 by unfolding the cycle formed by the edge labeled `b` at node 2; hence so G_1 and G_2 are bisimilar. The bisimulation relation $S : G_1.\mathrm{V} \times G_2.\mathrm{V}$ is $\{(1,1),(2,2),(2,2')\}$. Now suppose we insert for every `a`-labeled edge a reversed `a'`-labeled edge like in G_1' and G_2'. Notice the difference between outgoing edges from node 2 and those from node $2'$. In G_1', there is an outgoing edge `a'` from node 2. However, in G_2', there is no such edge from node $2'$. So G_1' and G_2' are not bisimilar anymore. The essential reason for this bisimulation genericity violation is that the reverse edge on a node is inserted on *incoming* edges while bisimulation is determined by *outgoing* edges. So bisimilar nodes are treated differently.

It would be interesting to combine λ_{FG} with the predicate of bisimilarity on graphs. Since (for ε-eliminable graphs) the bisimilarity is decidable, we can readily introduce such predicate in λ_{FG} and express queries that identify nodes in terms of their bisimilarity.

6. Related Work

Our structural recursion for ordered graphs is very much related to research on algebras of programming (Meijer et al. 1991; Bird and de Moor 1996; Hu et al. 2006), where structural recursions such as folds and catamorphisms are used to structure programs and systematically manipulate programs. In particular, our approach is influenced by the many attempts at defining structural recursions for specific graphs, such as graphs represented by trees with specific pointers (Hamana 2009; Dal Zilio et al. 2004; Oliveira and Cook 2012) and graphs represented by trees with embedded functions (Fegaras and Sheard 1996). However, they do not ensure all of the bisimulation genericity, terminating property, and finiteness preserving property, which are our original goals as explained in the introduction.

In the database community, structural recursion is an important primitive in database queries. Rewriting rules for optimization can be obtained by exploiting axiom of languages based on structural recursion. Although it works for various data models such as relations (Breazu-Tannen et al. 1991), nested collections (Wong 1994), unordered graphs (Buneman et al. 2000), and ordered trees (Robertson et al. 2009), structural recursion for querying ordered graphs has not been established yet.

As described in the introduction, our work was inspired by the structural recursion in UnCAL (Buneman et al. 2000) that is practically used for manipulating unordered graphs. We borrow from database community the idea of how to restrict a structural recursion for infinite regular trees so that a structural recursion for finite graphs preserves finiteness of graphs and becomes terminating, which are desirable properties for query languages. We solve the open problem of dealing with ordered graphs by providing a novel definition of bisimulation relation for ordered graphs. To enhance the expressive power to deal with transformation among children graphs, we extend the original structural recursion from $\mathbf{srec}(e)$ to $\mathbf{srec}(e, d)$ so that d can be used to combine results among children.

A lot of work has been devoted to efficient implementation of graph algorithms in lazy functional languages (Burton and Yang 1990; King and Launchbury 1995; Erwig 1997; Johnsson 1998). 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. In addition, unlike the above where graph equivalences are up to isomorphism, our graph equivalences are up to bisimilarity and all our transformations are guaranteed to be bisimulation generic.

Algebraic graph transformations (Ehrig et al. 2006) formalize graph transformations using categories in which objects are graphs and graph patterns to be matched, and matching be graph morphisms. A step of transformations is realized by a pushout complement followed by a pushout in the double-pushout approach. Category theory is elegantly used in the proofs and various constructions like transformation rule compositions. In contrast to our graph model, their graph model is based on graph isomorphism, and order between outgoing edges are not considered.

Picard and Matthes (2012) deal with node-labeled graphs using coinductive data types in Coq proof assistant. Although outgoing edges of their graphs are also ordered and graph equivalence based on bisimulation is considered, ε-edges are not considered.

Our decision procedure on ε-eliminability under finite-width in Section 3.3 is close to the eliminability of ε-transition cycles of weighted automata in Lombardy and Sakarovitch (2012). However, while eliminability of ε-transition cycle in the weighted automata is determined solely by the weight of the ε-edge (whether the weight is starable), we also consider the presence of proper branches. Note also that there is no notion of order between transitions in Lombardy and Sakarovitch (2012), so our ε-elimination cannot be derived from that of weighted automata.

The treatment for ε-edges in the current paper was inspired by (Jacobs 2010). Although there was no consideration of ε-edge itself in that study, the author showed that the trace semantics for some kinds of coalgebra induces iteration operators by forgetting the length of trace paths; the resulting iteration operators can be regarded as ε-elimination. The countable list monad CList is not treated in that paper (or in any literature to the best of our knowledge); it does not satisfy the assumption of the theorem in that paper. Since our definition of bisimilarity agrees with (strong) bisimilarity defined generally by coalgebra theory if graphs contain no ε-edges, Proposition 10 suggests that our bisimilarity for graphs having ε-edges can be equivalently defined by the combination of the usual strong bisimulation and ε-elimination. In this viewpoint, our essential contribution in Sections 3.1 and 3.2 is that we define and exploit CList and define the ε-elimination.

This paper is also related to our another submission (Asada et al. 2013), which also treats an extension of UnCAL. In the current paper, we provide a novel solution to the problem of querying over the ordered graphs, while Asada et al. (2013) shows that the solution can be *partially* generalized with less expressiveness of queries. More specifically, the two papers are technically independent in the following two points. First, though Asada et al. (2013) shows that branch patterns of graphs can be generalized by using monads, the generalization assumes the existence of a monad with an iteration operator. For ordered graph case, the definition of the monad CList with its iteration operator (i.e., the ε-elimination) is the very contribution of the current paper, and Asada et al. (2013) just use the result to include ordered graph case as an example of the generalization. The other point is that the current paper is not an instantiation of Asada et al. (2013); though the current paper extends the structural recursion with d for sibling transformations, it is not achieved in Asada et al. (2013). Also, the decidability of ε-eliminability is not generalized in Asada et al. (2013).

7. Conclusions and Future Work

We presented the first solution to the open problem of how to modify the graph model and structural recursion from unordered graphs to ordered ones and defined a new graph transformation language λ_{FG} for doing so. The key technical contributions are a novel definition of a bisimulation relation on ordered graphs having ε-edges and an extension of structural recursion with an operation for combining sibling results to achieve expressive power on the sibling dimension. We implemented λ_{FG} and showed two important optimization rules.

There are many interesting future paths of study. First, we should analyse structural recursion more thoroughly; for instance, we should examine the circumstances under which structural recursion is productive. A structural recursion is said to be productive, if it produces a finite ordered graph without ε-edges (i.e., $\mathrm{FG}\not\in$ defined in Section 3.3) for any input ordered graph without ε-edge. Second, following our previous work on bidirectionalizing UnCAL (Hidaka et al. 2010), we are very interested in developing a systematic way to bidirectionalize λ_{FG}.

Acknowledgments

We would like to thank Kazutaka Matsuda for various comments to the current and earlier versions of the paper. We also would like to thank the anonymous reviewers of the present and earlier versions for their thorough comments and constructive suggestions to improve the paper. The research was supported in part by the Grand-Challenging Project on the "Linguistic Foundation for Bidi-

rectional Model Transformation" of the National Institute of Informatics, and a Grant-in-Aid for Scientific Research for Encouragement of Young Scientists (B) No. 23700047, and a Grant-in-Aid for Scientific Research (B) No. 22300012.

References

K. Asada, S. Hidaka, H. Kato, Z. Hu, and K. Nakano. Parameterized graph transformation languages with monads. Technical Report GRACE-TR-2012-07, GRACE Center, National Institute of Informatics, 2012. `http://www.biglab.org/papers.html`.

K. Asada, S. Hidaka, H. Kato, Z. Hu, and K. Nakano. A parameterized graph transformation calculus for finite graphs with monadic branches, 2013. To appear in PPDP'13.

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

V. Breazu-Tannen, P. Buneman, and S. Naqvi. Structural recursion as a query language. In *Proc. of the Third International Workshop on Database Programming Languages(DBPL 91)*, pages 9–19, 1991.

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. POPL'04, pages 135–146, New York, NY, USA, 2004. ACM.

H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. Springer-Verlag, 2006.

M. Erwig. Functional programming with graphs. ICFP '97, pages 52–65, New York, NY, USA, 1997. ACM.

L. Fegaras and T. Sheard. Revisiting catamorphisms over datatypes with embedded functions. POPL '96, 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.

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.

M. Hamana. Initial algebra semantics for cyclic sharing structures. 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.

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.

S. Hidaka, Z. Hu, K. Inaba, H. Kato, K. Matsuda, K. Nakano, and I. Sasano. Marker-directed Optimization of UnCAL Graph Transformations. In *LOPSTR'11*, volume 7225 of *LNCS*, pages 123–138. Springer, 2012.

Z. Hu, H. Iwasaki, M. Takeichi, and A. Takano. Tupling calculation eliminates multiple data traversals. ICFP'97, pages 164–175, Amsterdam, The Netherlands, June 1997. ACM Press.

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.

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

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.

A. J. Kfoury and J. Tiuryn. Type reconstruction in finite-rank fragments of the polymorphic lambda-calculus (extended summary). In *LICS*, pages 2–11, 1990.

D. J. King and J. Launchbury. Structuring depth-first search algorithms in Haskell. POPL '95, pages 344–354, New York, 1995. ACM.

S. Lombardy and J. Sakarovitch. The removal of weighted ε-transitions. CIAA'12, pages 345–352. Springer-Verlag, 2012.

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.

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

B. C. Oliveira and W. R. Cook. Functional programming with structured graphs. ICFP '12, pages 77–88, New York, 2012. ACM.

C. Picard and R. Matthes. Permutations in Coinductive Graph Representation. In D. Pattinson and L. Schröder, editors, *Coalgebraic Methods in Computer Science*, volume 7399 of *LNCS*, pages 218–237. Springer, 2012.

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.

A. Takano and E. Meijer. Shortcut deforestation in calculational form. FPCA '95, pages 306–313, New York, 1995. ACM.

L. Wong. *Querying Nested Collections*. PhD thesis, Philadelphia, PA, USA, 1994.

A. Semantics of Graph Constructors

Figure 10 summarizes the semantic definitions of all graph constructors in λ_{FG}. In the definitions of $\underline{+\!\!+}$, $\oplus$, and @, we assume V_1 and V_2 to be disjoint, which is realized by taking copies of them.

The semantics should be easy to understand with the help of Figure 5 and the intuitive explanations given in Section 2.2.1.

B. Recursive Semantics with Output Markers

In Section 4.1, we gave recursive semantics by omitting output marker case. Throughout here, we consider general case with output markers. Hence, the typing rule is the one in Figure 4.

Now we will give both cases with and without PCC, whose definition will be given at the end. Before that, let us see more simple case, i.e., in addition to PCC, we assume that, for some monoid $(\odot, \iota_\odot)$ on $\mathbf{G}^Z_{Z\times\alpha+Z\times Y}$,

$$d = \mathbf{foldr}(\odot,\iota_\odot)\circ\mathbf{List}([p,q]) \quad : \mathbf{List}(\mathbf{G}^Z_{Z\times\alpha}+\mathbf{G}^Z_{Z\times Y}) \to \mathbf{G}^Z_{Z\times\alpha+Z\times Y}$$

where $p \stackrel{\text{def}}{=} (\text{-}) \mathbin{@} \lfloor \mathrm{in_l} \rfloor : \mathbf{G}^Z_{Z\times\alpha} \to \mathbf{G}^Z_{Z\times\alpha+Z\times Y}$ and $q \stackrel{\text{def}}{=} (\text{-}) \mathbin{@} \lfloor \mathrm{in_r} \rfloor : \mathbf{G}^Z_{Z\times Y} \to \mathbf{G}^Z_{Z\times\alpha+Z\times Y}$. Then, a structural recursion function $f \stackrel{\text{def}}{=} \mathbf{srec}(e,d)$ characterized by the following recursive semantics:

$$\begin{array}{lcl} f(\underline{[]}) & = & \iota_\odot \\ f(g_1 \mathbin{\underline{+\!\!+}} g_2) & = & f(g_1)\odot f(g_2) \\ f(\underline{[l:g]}) & = & e(l,g) \mathbin{@} f(g) \\ f(\underline{[\&y]}) & = & \lfloor \&z \mapsto (\&z,\&y) \rfloor \\ f(\&x := g) & = & (\oplus_{\&z\in Z}(\&z,\&x) := \underline{[}(\&z,\&)\underline{]}) \mathbin{@} f(g) \\ f(()) & = & () \\ f(g_1 \oplus g_2) & = & f(g_1)\oplus f(g_2). \end{array} \tag{7}$$

The fourth equation is the new one for the output marker case. The meaning should be quite easy to understand, and the following more general versions are essentially the same as this.

$$
\begin{array}{lcl}
\underline{[]} & = & (\{root\}, \{root \mapsto []\}, \{\& \mapsto root\}) \textbf{ where } root \text{ is a fresh node} \\
G_1 \mathbin{\underline{+\!\!+}} G_2 & = & \textbf{let } (V_1, B_1, I_1) = G_1;\ (V_2, B_2, I_2) = G_2 \\
 & & \quad \{\&x_1, \ldots, \&x_m\} = \ Dom(I_1)\ (= Dom(I_2)) \\
 & & \quad v_1, \ldots, v_m \text{ are fresh node identifiers} \\
 & & \quad B' = B_1 \cup B_2 \cup \\
 & & \qquad \{v_i \mapsto [\mathsf{Edge}(\varepsilon, I_1(\&x_i)), \mathsf{Edge}(\varepsilon, I_2(\&x_i))] \mid i = 1, \ldots, m\} \\
 & & \quad I'(\&x_i) = v_i \\
 & & \textbf{in } (V_1 \cup V_2 \cup \{v_1, \ldots, v_m\}, B', I') \\
\underline{[}l : G\underline{]} & = & \textbf{let } (V, B, I) = G \\
 & & \quad root \text{ is a fresh node} \\
 & & \textbf{in } (V \cup \{root\}, B \cup \{root \mapsto [\mathsf{Edge}(l, I(\&))]\}, \{\& \mapsto root\}) \\
\underline{[}\&y\underline{]} & = & (\{v\}, \{v \mapsto [\mathsf{Outm}(\&y)]\}, \{\& \mapsto v\}) \textbf{ where } v \text{ is a fresh node} \\
\&x := G & = & \textbf{let } (V, B, I) = G \textbf{ in } (V, B, \{\&x \mapsto I(\&)\}) \\
G_1 \oplus G_2 & = & \textbf{let } (V_1, B_1, I_1) = G_1;\ (V_2, B_2, I_2) = G_2 \\
 & & \textbf{in } (V_1 \cup V_2, B_1 \cup B_2, I_1 \cup I_2) \\
() & = & (\{\}, \{\}, \{\}) \\
G_1 \mathbin{@} G_2 & = & \textbf{let } (V_1, B_1, I_1) = G_1;\ (V_2, B_2, I_2) = G_2 \\
 & & \quad B'(v_1 (\in V_1)) = [\mathit{fillOutm}\ b \mid b \leftarrow B_1(v_1)] \\
 & & \qquad \textbf{where } \mathit{fillOutm}\ (\mathsf{Edge}(l', v')) = \mathsf{Edge}(l', v') \\
 & & \qquad\quad \mathit{fillOutm}\ (\mathsf{Outm}(\&y)) = \mathsf{Edge}(\varepsilon, I_2(\&y)) \\
 & & \quad B'(v_2 (\in V_2)) = B_2(v_2) \\
 & & \textbf{in } (V_1 \cup V_2, B', I_1) \\
\textbf{cycle}(G) & = & \textbf{let } (V, B, I) = G \\
 & & \quad B'(v) = [\ \mathit{cyclize}\ b \mid b \leftarrow B(v)] \\
 & & \qquad \textbf{where } \mathit{cyclize}\ (\mathsf{Edge}(l', v')) = \mathsf{Edge}(l', v') \\
 & & \qquad\quad \mathit{cyclize}\ (\mathsf{Outm}(\&y)) = \textbf{if } \&y \notin Dom(I) \textbf{ then } \mathsf{Outm}(\&y) \\
 & & \qquad\qquad\qquad\qquad \textbf{else } \mathsf{Edge}(\varepsilon, I(\&y)) \\
 & & \textbf{in } (V, B', I)
\end{array}
$$

Figure 10. Semantics of Graph Constructors

Now let us see another recursive semantics, with relaxing assumptions. For e and d satisfying PCC, we have the following recursive semantics.

$$
\begin{aligned}
& f(\underline{+\!\!+}_{i=1}^{n} \left\{ \begin{matrix} \underline{[}l_i : g_i\underline{]} \\ \underline{[}\&y_i\underline{]} \end{matrix} \right\}) \\
& = d_Y\big([\left\{ \begin{matrix} \mathrm{in_l}\big(e(l_i, g_i) \mathbin{@} f(g_i)\big) \\ \mathrm{in_r}\big(\lfloor\ \&z \mapsto (\&z, \&y_i)\rfloor\big) \end{matrix} \right\}]_{i=1}^{n}\big) \mathbin{@} \lfloor \nabla_{Z \times Y} \rfloor
\end{aligned} \tag{8}
$$

where $[x_i]_{i=1}^{n}$ is comprehension representation of a list $[x_1, ..., x_n]$. The above equation is read as follows: for each i on the left hand side, the content of {-} is upper or lower; then on the right hand side, for each i, according to the case on the left hand side, we think the upper or lower.

Compare the above with the following recursive semantics given in Section 4.1, where we omit the case of output markers:

$$
\begin{aligned}
& f([l_1 : g_1] \mathbin{\underline{+\!\!+}} ... \mathbin{\underline{+\!\!+}} [l_n : g_n]) \\
& = d_\emptyset\big([e(l_1, g_1) \mathbin{@} f(g_1), ..., e(l_n, g_n) \mathbin{@} f(g_n)]\big).
\end{aligned}
$$

In Section 4.1, as well as assuming Y to be $\emptyset$, we simplified the typing of d

$$
\begin{array}{ll}
\text{from} & d : \mathbf{List}(\mathbf{G}^Z_{Z\times\alpha} + \mathbf{G}^Z_{Z\times Y}) \to \mathbf{G}^Z_{Z\times\alpha + Z\times Y} \\
\text{to} & d : \mathbf{List}(\mathbf{G}^Z_{Z\times\alpha}) \to \mathbf{G}^Z_{Z\times\alpha}
\end{array}
$$

hence we did not need to use $\mathrm{in_l}$ and ∇ in Equation (8).

For general d without assuming PCC, we have the following recursive semantics.

$$
\begin{aligned}
& f(\underline{+\!\!+}_{i=1}^{n} \left\{ \begin{matrix} \underline{[}l_i : g_i\underline{]} \\ \underline{[}\&y_i\underline{]} \end{matrix} \right\}) \\
& = d_n\big([\left\{ \begin{matrix} \mathrm{in_l}\big(e(l_i, g_i) \mathbin{@} \lfloor \&z \mapsto (\&z, \&i)\rfloor\big) \\ \mathrm{in_r}\big(\lfloor\ \&z \mapsto (\&z, \&y_i)\rfloor\big) \end{matrix} \right\}]_{i=1}^{n}\big) \\
& \qquad \mathbin{@} \big((\textstyle\bigoplus_{i=1}^{n} \lfloor \mathrm{iso}_i \rfloor \mathbin{@} \left\{ \begin{matrix} f(g_i) \\ \underline{[]} \end{matrix} \right\}) \oplus \lfloor \mathrm{id}_{Z\times Y} \rfloor\big)
\end{aligned} \tag{9}
$$

where $\mathrm{iso}_i : Z \times \{i\} \cong Z$.

Production-Consumption Compatibility The PCC condition is defined as follows: e: $\mathbf{Label} \times \mathbf{G}_Y \to \mathbf{G}^Z_Z$ and d: $\mathbf{List}(\mathbf{G}^Z_{Z\times\alpha} + \mathbf{G}^Z_{Z\times Y}) \to \mathbf{G}^Z_{Z\times\alpha + Z\times Y}$ are called *production-consumption compatible (PCC)* if they satisfy the following: For any n, suppose that, for each $i \in n$, $l_i \in \mathcal{L}$, $G_i \in \mathbf{G}_Y$, and $G'_i \in \mathbf{G}^Z_{Z\times Y}$ are given, or $\&y \in Y$ is given. Then the following equation holds

$$
\begin{aligned}
& d_Y([\left\{ \begin{matrix} \mathrm{in_l}\big(e(l_i, G_i) \mathbin{@} G'_i\big) \\ \mathrm{in_r}\big(\lfloor \&z \mapsto (\&z, \&y)\rfloor\big) \end{matrix} \right\}]_{i=1}^{n}) \mathbin{@} \lfloor \nabla_{Z\times Y} \rfloor \\
& = d_n\big([\left\{ \begin{matrix} \mathrm{in_l}\big(e(l_i, G_i) \mathbin{@} \lfloor \&z \mapsto (\&z, i)\rfloor\big) \\ \mathrm{in_r}\big(\lfloor \&z \mapsto (\&z, \&y)\rfloor\big) \end{matrix} \right\}]_{i=1}^{n}\big) \\
& \qquad \mathbin{@} \big((\textstyle\bigoplus_{i=1}^{n} \lfloor \mathrm{iso}_i \rfloor \mathbin{@} \{ \begin{matrix} G'_i \\ \underline{[]} \end{matrix} \}) \oplus \lfloor \mathrm{id}_{Z\times Y} \rfloor\big)
\end{aligned}
$$

where $\nabla_{Z\times Y} : Z\times Y + Z\times Y \to Z\times Y$ and $\mathrm{iso}_i : Z \times \{i\} \cong Z$.