Bidirectional Transformation of Querying Ordered Graphs

Fei Yang

BASICS
Department of Computer Science and Engineering
Shanghai Jiao Tong University, P.R.China
iamyf@sjtu.edu.cn

1 Introduction

2 Preliminaries

Ordered graphs is a graph model in which the branchings are ordered. We start from introducing a formal definition for ordered graphs, and give a bisimilarity equivalence relation on the ordered graphs.

2.1 Ordered Graphs

The graphs we treated here are multi-rooted, directed, and edge-labeled graphs with order on outgoing edges. This graph model has three prominent features: *ϵ-edges*, *markers*, and *graph concatenation*. An ϵ-edges represents the a shortcut between two nodes, it is similar with the ϵ-transition in automata. Markers can be treated as interfaces that connect the nodes to other graphs. A node can be marked with *input* and *output markers*. Graph concatenation sequentially aligns two graphs in a given order.

The ordered graph model is formally defined as follows. We write $\mathcal{L}$ for the set of *labels* and $\mathcal{L}_\epsilon$ for $\mathcal{L} \cup \{\epsilon\}$. Let X and Y be finite sets of input and output markers; we add the prefix & for markers like $\&x$. An *ordered graph* G is defined by a triple (V, B, I), where

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

Here, every node in the range of I is called a root. Note that a root node may have incoming edges. However, we can convert it to a equivalent graph having no such incoming edges.

We denote a graph with the input marker set X and the output marker set Y by G^X_Y. Moreover, we write $\mathcal{G}^X_Y$ to represent the set of such graphs. Another assumption is that the graphs we are going to talk about are restricted to finite graphs, which means the

finiteness on the set of nodes. $\mathcal{G}^{cX}_{Y}$ is used to denote the set of ordered graph with a countble width. It is obvious that the set of finite ordered graphs $\mathcal{G}^{X}_{Y}$ is a subset of $\mathcal{G}^{cX}_{Y}$.

For the markers, we allow a graph to have multiple roots. When the graph is single-rooted, we often use & as the *default marker* to indicate the root and use $\mathcal{G}_Y$ ot denote $G^{\{\&\}}_{Y}$.

2.2 Graph Constructors

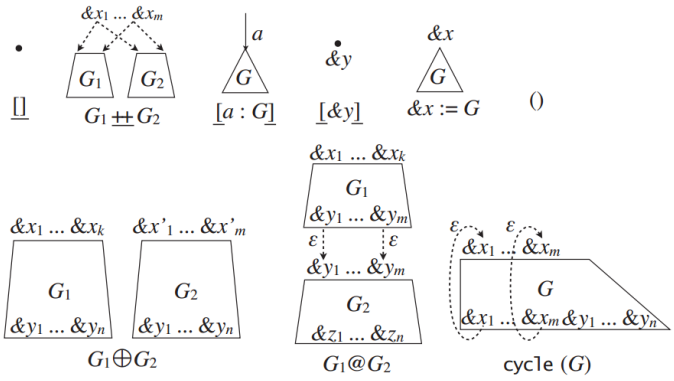

Fig. 1. Graph Constructors

Figure 1 sumarizes the nine costructors we used to describe arbitrary graphs.

$$
\begin{array}{lll}
G ::= & \underline{[]} & \{ \text{ single node graph } \} \\
| & G_1 \underline{+\!\!+} G_2 & \{ \text{ graph concatenation } \} \\
| & \underline{[a : G]} & \{ \text{ an edge pointing to a graph } \} \\
| & \underline{[\&y]} & \{ \text{ a node graph with an output marker } \} \\
| & \&x := G & \{ \text{ label he root node eith an input marker } \} \\
| & () & \{ \text{ empty graph } \} \\
| & G_1 \oplus G_2 & \{ \text{ disjoint graph union } \} \\
| & G_1 @ G_2 & \{ \text{ append of two graphs } \} \\
| & \textbf{cycle}(G) & \{ \text{ graph with cycles } \}
\end{array}
$$

The single node graph constructor $\underline{[]}$ construct a root-only graph. $G_1 \underline{+\!\!+} G_2$ adds two ϵ-edges from a new root to the roots of G_1 and G_2, respectively. $\underline{[a : G]}$ constructs a graph adding an a labeled edge point to the root of G. $\underline{[\&y]}$ constructs a single node graph with an output marker $\&y$. $\&x := G$ associates an input marker $\&x$ to the root node of G. $()$ constructs an empty graph with neither a node or an edge. Furthermore, $G_1 \oplus G_2$ constructs a graph by using a componentwise (V, B, I) union, $\underline{+\!\!+}$ differs from $\oplus$ in that $\underline{+\!\!+}$ unifies input nodes while $\oplus$ does not. $\oplus$ requires them to be identical. $G_1 @ G_2$

composes two graphs by connecting the output nodes of G_1 with the corresponding input nodes of G_2 with ϵ edges. **cycle**(G) connects the output nodes of G with the corresponding input nodes with ϵ edges, that will form cycles. The newly constructed nodes have unique identifiers. The formal definition of the sementics is given in λ_{FG}.

2.3 Proper Branches

In the definition of ordered graphs, the ϵ-edges are introduced to represent the shortcut between nodes. We need to relate two connected nodes by *proper branch*, where we can ignore the shortcuts. A *proper branch* of a node v is defined as a path from v going through zero or more ϵ-edges until it reaches a non-ϵ-edge or output marker.

Let $G = (V, B, I) \in \mathcal{G}^{cX}_{Y}$ and $v \in V$. The path starting from v

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

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. a non-ϵ-edge or an output marker. The set of all proper branches of v in G is denoted by $Pb(G, v)$.

Like the order on the branches, a total order is introduced on proper branches. Given two proper branches in an arbitrary graph, $p = (v \xrightarrow{\epsilon}_{i_0} v_1 \ldots \xrightarrow{\epsilon}_{i_{n-1}} v_n \rightarrow_{i_n})$, and $p' = (v \xrightarrow{\epsilon}_{i'_0} v'_1 \ldots \xrightarrow{\epsilon}_{i'_{n'-1}} v'_{n'} \rightarrow_{i'_n})$. Let their branch index sequences be $\tilde{p} \overset{def}{=} (i_0, \ldots, i_{n-1}, i_n)$ and $\tilde{p'} \overset{def}{=} (i'_0, \ldots, i'_{n'-1}, i'_{n'})$. We define $p \leqslant_{Pb} p' \overset{def}{\Longleftrightarrow} \tilde{p} \leqslant_l \tilde{p'}$ where $\leqslant_l$ is the lexicographical order between branch index sequences. This order plays an important roll in the definition of bisimilarity and the procedure of bidirectionalizing elimination of ϵ-edges.

For a given proper branch p, we use $p.last$ to denote the last step of the branch sequence, which is either a non-ϵ edge or a output marker.

2.4 Bisimilarity for Ordered Graphs

For the soundness of the graph model, we need an equivalence relation that can judge whether two arbitrary graphs are the same. At first glimpse, graph isomophism is the most straight forward approach. However, it is not necessarily that two graphs are isomophic when they behaves the same. Another option is language equivalence, which is focused on the language produced by the graphs, But unfortunately, it is proved to be undecidable in automata theory. Semantic equivalence is widely used in program verification to overcome the above difficulties. Bisimilarity is a kind of semantic equivalence, which is focused on the observable behavior of the graphs.

On unordered graphs, we use the notion of value equivalence (Bisimilarity) as the equivalence relation used in verfication. A similar bisimilarity on ordered graphs is given.

For two graphs $G = (V, B, I)$and $G' = (V', B', I')$ in $\mathcal{G}^{cX}_{Y}$, a relation R between V and V' is called a *bisimulation relation*, if for any vRv', there is an order isomorphism $f : (Pb(G, v), \leqslant_{Pb}) \rightarrow (Pb(G', v'), \leqslant_{Pb})$ satisfying he following order-preserving property: For any proper branch $p = (v \xrightarrow{\epsilon}_{i_0} v_1 \ldots \xrightarrow{\epsilon}_{i_{n-1}} v_n \rightarrow_{i_n}) \in Pb(G, v)$ with $f(p) = (v \xrightarrow{\epsilon}_{i'_0} v'_1 \ldots \xrightarrow{\epsilon}_{i'_{n'-1}} v'_{n'} \rightarrow_{i'_{n'}}) \in Pb(G', v')$, we have

– Edge Correspondence: if $B(v_n).i_n = \mathbf{Edge}(l, u)$ for some $l \in \mathcal{L}, u \in V$, then $\exists u' \in V'$ s.t. $B'(v'_{n'}).i'_{n'} = \mathbf{Edge}(l, u')$ and uRu',
– Marker Correspondence: if $B(v_n).i_n = \mathbf{Outm}(\&y)$ for some $\&y \in Y$, then $B'(v'_{n'}).i'_{n'} = \mathbf{Outm}(\&y)$.

Two graphs G and G' are *bisimilar* (denoted by $G \sim G'$) if there is a bisimulation relation R s.t. for every input marker $\&x \in X$, $I(\&x)RI'(\&x)$.

The notion of bisimilarity only consider the reachable part of a given graph. This means the unreachable parts are disregarded, i.e. two bisimilar garphs are still bisimilar if one adds subgraphs unreachable frome input nodes.

3 λ_{FG}: A Transformation Language for Ordered Graphs

3.1 The Core λ_{FG}

λ_{FG} is a transformation language for transforming (querying) ordered graphs. There is a formal semantics for constructing and querying an ordered graph in λ_{FG}. It is an extenstion of typed λ-calculus with graph constructors and list functions. It has a powerful feature of structural recursion. It is capable of querying with manipulating the sibling of nodes, which gives a great expresiveness power to the language. The syntex of λ_{FG} is given as below. It also has a system of strictly defined type inference rules.

$$
\begin{array}{lr}
\sigma ::= \sigma \to \sigma \mid \sigma + \sigma \mid \sigma \times \sigma & \{ \text{ function, coproduct, product types } \} \\
\quad \mid \mathbf{List}(\sigma) \mid \mathbf{Bool} & \{ \text{ list and boolean types } \} \\
\quad \mid \mathbf{Label} \mid \mathbf{G}^X_Y & \{ \text{ label and graph types } \}
\end{array}
$$

$$
\begin{array}{lr}
e ::= x \mid \lambda x.e \mid e\, e \mid \mathbf{case}\, e\, \mathbf{of}\, \mathbf{in_l}(x) \to e\, \mathbf{or}\, \mathbf{in_r}(y) \to e & \\
\quad \mid \mathbf{in_l} e \mid \mathbf{in_r} e \mid (e, e) \mid \mathbf{pr_l} e \mid \mathbf{pr_r} e & \{ \text{ terms of lambda calculus } \} \\
\quad \mid \mathbf{nil} \mid \mathbf{cons}(e, e) \mid \mathbf{foldr}(e, e) \mid \ldots & \{ \text{ functions for lists } \} \\
\quad \mid \mathbf{if}\, e\, \mathbf{then}\, e\, \mathbf{else}\, e & \{ \text{ conditional } \} \\
\quad \mid a|e = e & \{ \text{ labels } (a \in \mathcal{L}) \text{ and label equality } \} \\
\quad \mid \underline{[]} \mid e \underline{+\!\!+} e \mid \underline{[e : e]} \mid \underline{[\&y]} \mid \&x := e \mid () \mid e \oplus e & \\
\quad \mid e@e \mid \mathbf{cycle}(e) & \{ \text{ graph constructors } \} \\
\quad \mid \mathbf{isEmpty}(e) & \{ \text{ graph emptiness checking } \} \\
\quad \mid \mathbf{srec}(e, e) & \{ \text{ structural recursion functions } \}
\end{array}
$$

Here we focus on the explaination of *structural recursion*, which is a powerful graph transformation mechanism. It provides the capability to descirbe the queries and transformations that garantees the termination of the computation and preserves the finiteness.

In general the structure recursion is written of the form $\mathbf{srec}(e, d)$, wherer e can be considered as the body function applied on the labeled edges and d is a map computation on nodes. For a simple case, we can consider $d = \mathbf{foldr}(\odot, \iota_\odot)$ for some monoid $(\odot, \iota_\odot)$ on $\mathbf{G}^Z_{Z \times \alpha}$. The structural recursion function $f \stackrel{def}{=} \mathbf{srec}(e, d)$ satisfies the follow-

ing equations (herer, equality means bisimilarity):

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

Example 1. This example shows how to manipulate edges of the graph and change its shape by structural recursion, The following function $a2d\_xc$ replaces all labels a with d and contracts c-labeled edges.

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

Compared with the structural recurcion in UnCAL, structural recurtion in λ_{FG} can deal with orered graphs and computations on the sibling dimension of graphs. This ability is given by the function of d.

3.2 Bulk Semantics of Structural Recursion

The structural recursion of λ_{FG} has two equivalent semantics. One is *recursive semantics*. It defines the function behavior and the program transformation/optimization recursively. It appears to be more concise and is useful for reasoning. Another one is *bulk semantics*. By allowing ϵ-edges, we can evaluate a structural recursion in a *bulk* manner. The bulk semantics consider the transformation on a whole. It is useful in parallel computation and the proof of termination and finiteness-preserving property. In the construction of bidirectional semantics, we will also apply the bidirectionlization on bulk semantics.

Before the introduction of bulk semantics, let us consider the type of the structural recursion function $\mathbf{srec}(e, d)$ in advance. A type inference rule is given below.

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

Here, e is the body function of the transformation on edges. it performs a similar role as the body function in the unordered version. It maps the a subgraph of a labeled edge into the result graph. For the d function, it works as a rearrangement function over the lists. It maps the each lists by manipulate the order of the branches. In the bulk semantics, the evaluation can be applied in parallel. It mainly involves three steps.

1. Map computation on edges with e: the function e is applied on each labeled edge, yield a set of intermediate result graphs for e. The set graphs will be rearranged in the next step.

2. Map computation on nodes with d: the function d is applied on each nodes. The previous set of graphs are merged by the rearrangement measure defined in d, leads to a set of merged graphs for each node.
3. Groups new graphs with ϵ-edges: to get the final result, the graphs for each node need to be grouped by ϵ-edges. The ϵ-edges are added according to the input and output markers of the previous result.

4 An Overview of Bidirectional Transformation

In UnCAL, there is a framework that could bidirectionalize unordered graph transformation. It consists of a two stage bidirectionalizing strategy and it could solve the problem that the graphs may contain shared nodes or cycles. This framework provides us a some heuristic idea to do the bidirectionalizing on λ_{FG}.

The challenging parts of bidirectionalizing on λ_{FG} comes from the d function in the structural recursion. It leads to a three step bulk semantics, which would make it hard to trace back from the view. Further, the d function only accepts graphs without ϵ-edges. Thus, an ϵ-elimination procedure has to be applied before the structural recursion for an arbitrary graph. Also, the ϵ-edge should be eliminated before it is presented to the customer. Moreover, the order of the branchings must be treated carefully in our bidirectionalization.

4.1 Bidirectional Properties

The goal in bidirectionalizing λ_{FG} is to approach the bidirectionalization in ordered graphs by providing a bidirectional semantics in λ_{FG}. The semantics should have bidirectional properties defined in the previous work in UnCAL. Let $\mathscr{F}[\![e]\!]\rho$ denote a forward evaluation (get) of expression e under environment ρ to produce a view, and $\mathscr{B}[\![e]\!](\rho, G')$ denote a backward evaluation (put) of expression e under environment ρ to reflect a possibly modified view G' to the source by computing an updated environment. An *environment* ρ is a mapping with a form $\{\$x \mapsto X, \ldots\}$ where X is a graph G or a label l, and $\$x$ is a variable. The transformation need to satisfy the following two important properties:

$$\frac{\mathscr{F}[\![e]\!]\rho = G}{\mathscr{B}[\![e]\!](\rho, G) = \rho}(GETPUT) \qquad \frac{\mathscr{B}[\![e]\!](\rho, G') = \rho' \mathscr{F}[\![e]\!]\rho' = G''}{\mathscr{B}[\![e]\!](\rho, G'') = \rho'}(WPUTGET)$$

The (GETPUT) property states that unchanged view G should give no change on the environment ρ in the backward evaluation, while the (WPUTGET) property states that the modified view G' and the view obtained by backward evaluation followed by forward evaluation may differ, but both view have the same effect on the original source if backward evaluation is applied. A pare of forward and backward evaluation is *well-behaved* if it satisfies (GETPUT) and (WPUTGET) proties. In the rest of the paper, the forward and backward evaluations will be presented for λ_{FG}, and the following theorem holds.

Theorem 1 (Well-behavedness). *The proposed forward and backward evaluations are well-behaved, provided their evaluations succeed.*

According to the (WPUTGET) property, given a certain forward evaluation function, there could be more than one backward evaluation that would satisfy the well-behavedness property. When we perform the modification on the view, the most reasonable choice is to reflet it on the corresponding part on the source. This is the *location correspondence* property. Moreover, the modification on the view graph could be categorised into three types: in place updates, edge deletion and edge insertion. In order to maintain the well-behavedness between the updated source and the modified view, we also need to apply some operation on the source. A reasonable assumption is that a certain type of modification on the view yields the same type of operation on the source, which could be called the *operation correspondence* property.

Theorem 2 (Rationality). *The proposed bidirectional transformation on λ_{FG}, satisfies both location and operation correspondence property. We call it a rational transformation.*

The rationality property is also important as it gives us a discriptive goal in designing the bidirectionalizing the transformation. The goal is to let the updates on the source to meet the intention of the changing on the view.

4.2 Three-Stage Bidirectionalization

In an arbitrary transformation, there could be a big gap between the source and view graoh. It makes it hard to reflect changes on the view to the source. The basic idea to divide the forward evaluation into three stages, each one could be bidirectionalized.

- Stage 1: Elimination of ϵ-edges on the source, to produce a simple source that we can apply bulk semantics on it.
- Stage 2: Forward evaluation using bulk semantics. This stage may create some new ϵ-edges, so that the output graph will have a similar shape to the input graph. Appropriate trace information might be appended locally on the nodes in the result graph.
- Stage 3: Elimination of ϵ-edges to produce a usual view.

In our bidirectional transformation framework, we will propose a general bidirectionalized ϵ-edge elimination procedure for both Stage 1 and Stage 3. For Stage 2, a bidirectionalized semantics for λ_{FG} is provided in this paper. Thus, the whole transformation is bidirectionlized by a inductively defined procedure.

5 Bidirectionalizing ϵ-elimination procedure

In the previous work of λ_{FG}, the input graph of the d function in structural recursion does not accept any ϵ-edges. However, we can not safely assume that an arbitrary input graph does not contain any ϵ-edges. Further, after applying the transformation via bulk semantics, some ϵ-edges might be introduced to maintain the structure of the result

graph. In most applications, ϵ-edges are assumed to be unobservable. Thus, we wish to produce a view without any ϵ to the users. In this seciton, a bidirectionalized ϵ-elimination procedure is proposed.

First, there is an effective procedure that whether the ϵ-elimination could be applied successfully on a finte ordered graph G. The details could be found in the paper of λ_{FG}.

Lemma 1. *Given an finite ordered graph G possibly with ϵ-edges, there is an effective procedure to decide whether there exists an ordered graph G', s.t. $G \sim G'$ and has neither ϵ-edges nor infinite width. If yes, produce such G'.*

The idea of the ϵ-elimination procedure is to use labeled edges to connect the shortcut between proper branches, the procedure is defined as below. Note that we use a list of integers to represent the total orders over the branches and proper branches on a node.

Definition 1. *For a graph $G = (V, B, I) \in \mathcal{G}^{cX}_{Y}$, the ϵ-elimination ϵ-elim(G) of G is a graph $(V, B'I) \in \mathcal{G}^{cX}_{Y}$ where $|B'(v)| \stackrel{def}{=} Pb(G, v)$, and $B'(v).\tilde{p} \stackrel{def}{=} Pb(G, v).\tilde{p}.last$ for $p = (v \xrightarrow{\epsilon}_{i_0} \ldots v_n \rightarrow_{i_n})$ in $B'(v)$.*

Next we will introduce how to reflect view updates within ϵ-elim(G) to the source graph G. We will use the correspondence between the proper branches in the source graph and the branches in the view graph.

1. For the inplace-updates on an arbitrary edge $B'(v).\tilde{p} \in \epsilon$-elim$(G)$, the corresponding labeled edge on the view is $Pb(G, v).\tilde{p}.last$. We could apply the same update operation on this edge.
2. For the edge deletion on an arbitrary edge $B'(v).\tilde{p} \in \epsilon$-elim$(G)$, we update the source graph by delete the correcponding edge $Pb(G, v).\tilde{p}.last$.
3. For the edge insertion, since the set of nodes V is the same on the view and the source. Thus, we just need to insert the edge to the same place on the source.

After we have done the update operation on the source, an ϵ-elimination procedure need to be applied again on the updated source. The reason is that, the updated labeled edge may be duplicated in different proper branches. Thus, the corresponding edge of these proper branches on the view graph also need to be updated to mainain the consistency between the source and the view. For inplace updates and edge deletion, the corresponding edge is deleted on the view, and for edge insertion, the edge is added according to the newly added proper branches. Therefore, the whole procedure would satisfy the well-behavedness. Moreover, the rationality is also infered from this constructive bidirectionalizing procedure.

Lemma 2. *The proposed bidirectionalized ϵ-elimination procedure satisfy the well-behavedness and the rationality property.*

6 Traceble Forward Evaluation

Practically, a λ_{FG} expression usually specifies a forward transformation that maps a source ordered graph (possibly a database or an XML file) to a view graph. A backward transformation for this procedure specifies how to reflect view updates to the source graph. However, the structural information kept by ϵ-edges will not appear explicitely in the view graph. Moreover, the newly created parts within the transformation procedure also make it hard to detect the generated source for everything in the view graph. In this sence, some appropriate trace information should be added to the view, to make the view more informative, in other words, *traceable*. In this section, we will enrich the original semantics of λ_{FG} to make it produces traceable view after forward transformation.

6.1 Local Trace Information

Recall the semantics in λ_{FG}, a view is obtained by evaluating a λ_{FG} expression with an ordered graph. Every node in the view graph is either a graph obtained from the source graph, or a node constructed by te λ_{FG} expression, except the node generated through a structural recursion(in bulk semantics). For an arbitrary structural recursion function $\mathbf{srec}(e_1, d)(e_2)$, the input of the body function e_1 is evaluated binding variables from a part of evaluated result in e_2; and the input of the rearrangement d function is evaluated by binding variables from the result in e_1. So the node in the result graph may originate from not only the whole *srec* expression but also a sub-expression in e_2. Thus, a more complicated inductively defined trace information should be added on the nodes on the result graph.

Here, we use the a *traceable view* in which each node has information for tracing its origin. This locally added information, called $TraceID$, is defined by

$$
\begin{aligned}
TraceID \coloneqq\ & SrcID \\
\mid\ & Code\ Pos\ Marker \\
\mid\ & RecN\ Pos\ TraceID\ Marker \\
\mid\ & RecE\ Pos\ TraceID\ TraceID\ Num \\
\mid\ & RecM\ Pos\ Marker\ TraceID\ Num \\
\mid\ & RecD\ Pos\ TraceID\ TraceID
\end{aligned}
$$

Where $SrcID$ ranges over the identifiers uniquely assigned to all nodes in the source graph, Pos ranges over code positions in the λ_{FG} expressions, $Marker$ ranges over input/output makers, and Num ranges over the lists of integers, which represent the order of the branches.

Now we briefly expain the meaning of each $TraceID$. $SrcID$ means the node is generated from the corresponding one in the source. $Code\ Pos\ Marker$ is the information for the nodes created by the λ_{FG} expreesion on the marked code position. *RecN*, *RecE*, *RecM*, *RecD* are four kinds of trace information used in bidirectionalizing structural recursion $\mathbf{srec}(e, d)$. *RecN RecE RecM* represent the node in the intermediate result after e is applied, which are generated from the nodes/edges/markers respectively, and *RecD* information is added when the rearrangements on the lists are applied by d.

6.2 Enriched Forward Semantics

Now we will define the enriched forward semantics of λ_{FG} that could generate a traceable view. Recall the core of λ_{FG} language introduced in 3.1, the tracing information is recorded when a node is created. Let e^p denote an λ_{FG} subexpression e at code position p. We write $\rho(\$x)$ for G when $(\$x \mapsto G) \in \rho$, and $e\rho$ means the variable substitution using ρ in the expression e. We inductively define the enriched forward semantics $\mathscr{F}[\![e^p]\!]\rho$ for each λ_{FG} construct of e.

In addition, we replace the composition of markers in $X \times Z$ by a monoid $(.,\&)$, with $\&$ as the identity, i.e., $\&.\&x = \&x.\& = \&x$. In particular, $\{\&\}.Z = Z.\{\&\} = Z$ for any Z. We call "." the Skolem function.

Graph Constructor Expressions The semantics of graph constructor expressions is straightforward according to the construction in Fig1. For instance, the enriched semantic of single node graph constructor is,

$\mathscr{F}[\![\underline{[]}^p]\!]\rho = (\{Code\ p\}, \{Code\ p \mapsto []\}, \{\& \mapsto Code\ p\})$

Note that the code position is added in the node, which infers that it is generated by the expression of in position p. Moreover, the branch set is nonempty despite there is no edge in the graph.

As another example, the semantics for the expression $e_1 \underline{+\!\!+} e_2$ is defined below. Note that there is no such combinator in unordered version for that it remains the order of the graphs when it links the input markers.

$\mathscr{F}[\![(e_1 \underline{+\!\!+} e_2)^p]\!]\rho = \mathscr{F}[\![e_1]\!]\rho \underline{+\!\!+}^p \mathscr{F}[\![e_2]\!]\rho$
let $G_1 \underline{+\!\!+}^p G_2 = (V_1 \cup V_2 \cup V', B_1 \cup B_2 \cup B', I')$
where $(V_1, B_1, I_1) = G_1,\ (V_2, B_2, I_2) = G_2$
$M = \mathbf{inMarker}(G_1) = \mathbf{inMarker}(G_2)$
$V' = \{Code\ p\ \&m \mid \&m \in M\}$
$B' = \{Code\ p\ \&m \mapsto [\mathbf{Edge}(\epsilon, I_1(\&m)), \mathbf{Edge}(\epsilon, I_2(\&m))] \mid \&m \in M\}$
$I' = \{\&m \mapsto Code\ p\ \&m \mid \&m \in M\}$

where $\underline{+\!\!+}^p$ is a union operator for two graphs concerning position p. We write $\mathbf{inMarker}(G)$ and $\mathbf{outMarker}(G)$ to denote the set of input and output markers in a graph G, respectively.

$e_1 \oplus e_2$ is a componentwise union operator like $\underline{+\!\!+}$, exept that no ϵ-edges are involved.

$\mathscr{F}[\![(e_1 \oplus e_2)^p]\!]\rho = \mathscr{F}[\![e_1]\!]\rho \oplus \mathscr{F}[\![e_2]\!]\rho$
let $G_1 \oplus G_2 = (V_1 \cup V_2, B_1 \cup B_2, I_1 \cup I_2)$
where $(V_1, B_1, I_1) = G_1; (V_2, B_2, I_2) = G_2$

For the empty graph constructor and constant marker graph, it is easy to define

$\mathscr{F}[\![()^p]\!]\rho = (\{\}, \{\}, \{\})$

$\mathscr{F}[\![\underline{[\&m]}^p]\!]\rho = (\{Code\ p\}, \{Code\ p \mapsto [\mathbf{Out}m(\&m)]\}, \{\& \mapsto Code\ p\})$

The next two constructors append labeled edges and markers with a graph respectively.

$\mathscr{F}[\![\underline{[l : e]}^p]\!]\rho = (\{Code\ p\} \cup V, \{Code\ p \mapsto [(l\rho, I(\&))]\} \cup B, \{\& \mapsto Code\ p\})$
where $(V, B, I) = \mathscr{F}[\![e]\!]\rho$

TraceID is generated for the newly constructed node.

$\mathscr{F}[\![(\&m := e)^p]\!]\rho = (\&m := \mathscr{F}[\![e]\!]\rho)$

let $(\&m := G) = (V, B, I')$

where $(V, B, I) = G$

$I' = \{\&m.\&x \mapsto v \mid (\&x \mapsto v) \in I\}$

Here "." is the Skolem function introduced before.

$e_1@e_2$ appends two graphs by connecting the output nodes of the left operand and corresponding input nodes of the right operand with ϵ-edges.

$\mathscr{F}[\![(e_1@e_2)^p]\!]\rho = \mathscr{F}[\![e_1]\!]\rho \ @^p \mathscr{F}[\![e_2]\!]\rho$

let $G_1@^pG_2 = (V_1 \cup V_2, B_1' \cup B_2, I_1)$

where $(V_1, B_1, I_1) = G_1,\ (V_2, B_2, I_2) = G_2$

$$B_1' = \{u \mapsto \begin{bmatrix} x.i = \mathbf{Edge}(l, v) & \Rightarrow x.i \\ = \mathbf{Outm}(\&m) & \Rightarrow (\epsilon, I_2(\&m)) \end{bmatrix}_{i \in |x|} \mid (u \mapsto x) \in B_1\}$$

cycle(e) is defined as follows

$\mathscr{F}[\![(\mathbf{cycle}(e))^p]\!]\rho = \mathbf{cycle}^p(\mathscr{F}[\![e]\!]\rho)$

let $\mathbf{cycle}^p(G) = (V, B', I)$

where $(V, B, I) = G$

$$B' = \{u \mapsto \begin{bmatrix} x.i = \mathbf{Edge}(l, v) & & \Rightarrow x.i \\ = \mathbf{Outm}(\&m) & \wedge \&m \in dom(I) & \Rightarrow \mathbf{Edge}(\epsilon, I(\&m)) \\ = \mathbf{Outm}(\&m) & \wedge \&m \notin dom(I) & \Rightarrow x.i \end{bmatrix}_{i \in |x|} \mid (u \mapsto x) \in B\}$$

Labels

Variables

Condition

Emptiness Checking

Function on Lists

Structural Recursion

7 Backward Evaluation

7.1 Reflection of Inplace Updates

7.2 Reflection of Edge Deletion

7.3 Reflection of Edge Insertion

8 Conclusion