ISSN 1884-0760

GRACE TECHNICAL REPORTS

Graph Transformation as Graph Reduction FUnCAL: A Functional Reformulation of Graph-Transformation Language UnCAL

Kazutaka Matsuda Kazuyuki Asada

(Communicated by Zhenjiang Hu)

GRACE-TR 2015-01 January 2015

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

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

Graph Transformation as Graph Reduction

FUnCAL: A Functional Reformulation of Graph-Transformation Language UnCAL

Kazutaka Matsuda
The University of Tokyo
kztk@is.s.u-tokyo.ac.jp

Kazuyuki Asada
The University of Tokyo
asada@kb.is.s.u-tokyo.ac.jp

Abstract

A large amount of graph-structured data are widely used, including biological database, XML with IDREFs, WWW, and UML diagrams in software engineering. UnCAL, proposed by Buneman et al. from the database community, is a language designed for *graph transformations*, i.e., extracting a subpart of a graph data and converting it to a suitable form, which often also has a graph structure. A distinguished feature of UnCAL is its semantics that respects *bisimulation* on graphs; this enables us to reason about UnCAL graph transformations as recursive functions, which is useful for verification as well as optimization.

However, despite of this similarity of UnCAL to functional languages, there is still a gap to apply the program-manipulation techniques studied in the programming language literature directly to UnCAL programs, due to some special features in UnCAL, especially *markers*.

In this paper, first, we give a translation from UnCAL programs to functional ones by emulating markers by tuples and λ-abstractions, so that we can reason about UnCAL programs through functional ones. Thanks to the translation, we can import several verification results designed for functional programs to the UnCAL transformations. Second, to optimize UnCAL graph transformations as functional programs, we give a memoized lazy semantics and a type system so that a well-typed functional program terminates and results in a finite graph, under the semantics; that is, well-typed functional programs are graph transformations. Thanks to the semantics and the type system, we can optimize a translated functional program freely as long as the optimization keeps typability, and execute it as a graph transformation.

Keywords Graph Transformation, Functional Languages, Lazy Evaluation, Bisimulation, Regular Trees, Termination, Memoization

1. Introduction

A large amount of graph-structured data are widely used, including biological information, XML with IDREFs, WWW, UML diagrams in software engineering [18], and Object Exchange Model (OEM) for exchanging arbitrary database structures [37]. In such circumstances, several languages, such as UnQL/UnCAL [8], Lorel [1], GraphLog [9], have been proposed mainly from the database community for *graph transformation* or querying over such graph-structured data—extracting a subpart of a graph and converting it to some suitable form—similarly to what XQuery does for XMLs. For example, we want to extract the orders by "Tanaka" from the graph in Figure 1 containing information of customers and their orders to obtain the graph in Figure 2.

UnCAL is a prominent language designed for graph transformations [8]. Among its other nice features such as termination guarantee and efficient execution by ε-edges [8], the most characteristic feature of UnCAL is its semantics that respects *bisimulation*, under which a graph and the infinite tree obtained by unfolding sharings and cycles are equivalent. Thanks to the bisimulation-respecting semantics, UnCAL supports functional-programming-style reasoning: one can reason about UnCAL graph transformations as recursive functions that generate infinite trees, which is useful for verification [24] as well as optimization [8, 21].

However, despite of this similarity of UnCAL to functional languages, there is still a gap between UnCAL programs and functional ones, which will be explained in more detail in Section 1.1. Due to the gap, it is hard to apply program-manipulation techniques studied in the programming language literature directly to UnCAL programs. This is unfortunate to both communities; the database community cannot enjoy well-studied programming-language techniques, and the programming-language community loses chances to contribute to the other community. Actually, several methods have been proposed for UnCAL while there already have been similar methods in the programming language literature. For example, the key technique in the optimization in [8, 21] is quite similar to the classic fold-fusion [32].

The purpose of this paper is to fill the gap between UnCAL and usual functional languages so that we can directly apply program-manipulation techniques studied in the programming-language community to the graph transformation problem. Specifically, in this paper, we give a translation from UnCAL programs to functional ones so that we can reason about, manipulate and execute UnCAL programs as functional ones.

1.1 Problem and Observation

The gap between UnCAL and usual functional languages, or what prevent us from directly importing techniques studied in the programming language community, is *markers* used to connect two graphs and to construct cycles. We have to cope with markers to seamlessly import functional results.

There are two usages of markers: *input* and *output*. Roughly speaking, input markers are names for multiple-roots and output markers are names for holes. UnCAL also has expressions that connect nodes indicated by input markers (*input nodes*) and those indicated by output markers (*output nodes*) of the same names.

Let us review how markers are used in UnCAL. First, we explain UnCAL expressions that do not use any markers. Without

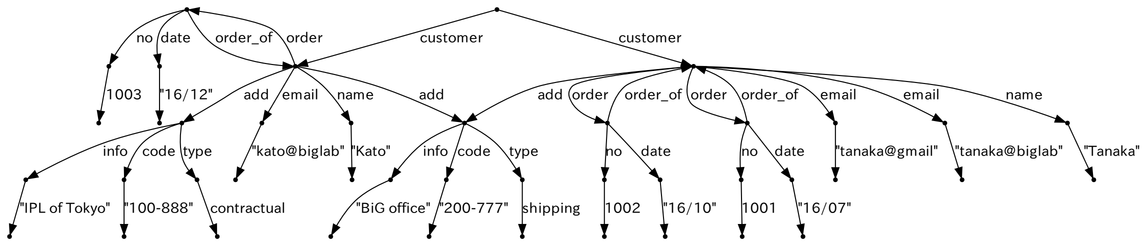


Figure 1. A graph data containing order information in customer-oriented representation.

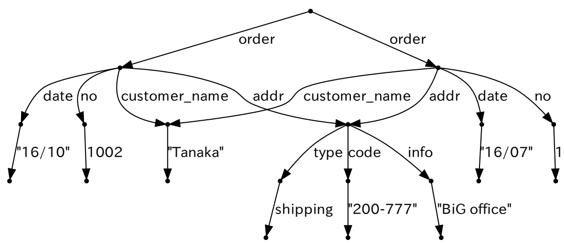


Figure 2. A graph data obtained from Figure 1 by extracting orders by "Tanaka" in order-oriented representation.

markers, graphs in UnCAL are similar to records, as below.

$$\{\mathtt{name} : \mathtt{Alice}, \mathtt{email} : \mathtt{alice}\}$$

Markers are used as an interface for connecting other graphs. In the following graph, we can connect something to output node $\&x$.

$$\{\mathtt{name} : \mathtt{Alice}, \mathtt{friend} : \&x\}$$

A graph to be substituted to the output node must have the corresponding input maker, which can be assigned by $\triangleright$ as below.

$$\&x \triangleright \{\mathtt{name} : \mathtt{Bob}, \mathtt{friend} : \&y\}$$

Then, we can connect the two graphs by @; for example, by writing

$$\begin{array}{l}\{\mathtt{name} : \mathtt{Alice}, \mathtt{friend} : \&x\} \\ \ @\ (\&x \triangleright \{\mathtt{name} : \mathtt{Bob}, \mathtt{friend} : \&y\})\end{array}$$

we get the following graph.

$$\{\mathtt{name} : \mathtt{Alice}, \mathtt{friend} : \{\mathtt{name} : \mathtt{Bob}, \mathtt{friend} : \&y\}\}$$

Cyclic graphs can be constructed by $\mathbf{cycle}(g)$ that connects input nodes and output nodes in g, as follows.

$$\mathbf{cycle}\left(\&y \triangleright \left(\begin{array}{l}\{\mathtt{name} : \mathtt{Alice}, \mathtt{friend} : \&x\} \\ @\ (\&x \triangleright \{\mathtt{name} : \mathtt{Bob}, \mathtt{friend} : \&y\})\end{array}\right)\right)$$

Then, we obtain a cyclic graph that represents `Alice` and `Bob` are friends of each other.

Graph transformations are written by using **srec**, a structural recursion on graphs. Thanks to the bisimulation-respecting semantics, **srec** can be understood as if it were defined recursively as:

$$\begin{array}{l}\mathbf{srec}(e)(\{\mathtt{a}_1 : G_1, \dots, \mathtt{a}_n : G_n\}) = \\ \quad (e(\mathtt{a}_1, G_1)\ @\ \mathbf{srec}(e)(G_1)) \cup \dots \cup (e(\mathtt{a}_n, G_n)\ @\ \mathbf{srec}(e)(G_n))\end{array}$$

Here, $\cup$ is the record concatenation; actually $\{\mathtt{x} : s, \mathtt{y} : t\}$ in UnCAL is a shorthand notation for $\{\mathtt{x} : s\} \cup \{\mathtt{y} : t\}$. For example, the following UnCAL expression returns people named `Bob` in db where db is a variable that stores a record of the form of $\{\mathtt{person} : p_1, \dots, \mathtt{person} : p_n\}$.

$$\begin{array}{l}\mathbf{srec}(\lambda(\_, p). \\ \ \mathbf{srec}(\lambda(l, n).\&r \triangleright \\ \ \ \mathbf{if}\ l = \mathtt{name}\ \mathbf{then} \\ \ \ \ \mathbf{srec}(\lambda(l', \_).\mathbf{if}\ l'{=}\mathtt{Bob}\ \mathbf{then}\ \{\mathtt{person} : p\}\ \mathbf{else}\ \{\})(n) \cup \&r \\ \ \ \mathbf{else} \\ \ \ \ \&r)(p))(db)\end{array}$$

The output node $\&r$ represents the result of the recursive call of the second-outermost **srec**.

One might notice that these behaviors of input/output markers, @, and **cycle** can be emulated by λ-abstractions and **letrec**. For example, the UnCAL expression above that constructs the cyclic graph can be written as below.

$$\begin{array}{l}\mathbf{letrec}\ y = (\lambda x. \{\mathtt{name} : \mathtt{Alice}, \mathtt{friend} : x\}) \\ \qquad\qquad \{\mathtt{name} : \mathtt{Both}, \mathtt{friend} : y\}\ \mathbf{in}\ y\end{array}$$

Also, one might notice that the behavior of **srec** can be expressed by a paramorphism [31] *para* that behaves like:

$$\begin{array}{l}para\ f\ \{\mathtt{a}_1 : G_1, \dots, \mathtt{a}_n : G_n\} \\ \ = (f\ \mathtt{a}_1\ G_1\ (para\ f\ G_1)) \cup \dots \cup (f\ \mathtt{a}_n\ G_n\ (para\ f\ G_n))\end{array}$$

It would seem that reasoning and execution of UnCAL programs as functional ones would look straightforward.

However, the straightforward translation is not enough because the translation can map terminating UnCAL expressions to nonterminating ones. This is problematic if we apply optimization techniques such as fusion [32] to UnCAL programs because we may not execute optimized translated programs, although the translation still is useful in reasoning of UnCAL programs. For example, the translation converts $\mathbf{cycle}(\&x \triangleright \&x)$, which results in a singleton graph in UnCAL, to $\mathbf{letrec}\ x = x\ \mathbf{in}\ x$, which leads to an infinite loop in usual languages. Although the expression $\mathbf{cycle}(\&x \triangleright \&x)$ itself is rarely seen in practice, a similar problem arises when we write graph transformations by **srec**. For example, let us consider the following UnCAL expression that eliminates all the edges from db and thus returns a singleton graph for any db.

$$\mathbf{srec}(\lambda(\_, \_).\&r \triangleright \&r)(db)$$

The transformation can be seen as a simplified version of the above transformation that searches `Bob`, in the sense that it models the behavior of the second-outermost **srec** of the transformation when it is applied to a graph with no `name`s. Here comes a problem. The behavior of the transformation differs after the translation if we apply it to a cyclic graph like that obtained by $\mathbf{cycle}(\&x \triangleright \{\mathtt{a} : \&x\})$. The UnCAL expression

$$\mathbf{srec}(\lambda(\_, \_).\&r \triangleright \&r)(\mathbf{cycle}(\&x \triangleright \{\mathtt{a} : \&x\}))$$

terminates and returns a singleton graph while the corresponding functional program

$$para\ (\lambda\_.\lambda\_.\lambda r.r)\ (\mathbf{letrec}\ x = \{\mathtt{a} : x\}\ \mathbf{in}\ x)$$

goes into an infinite loop.

Another but related issue is that we want to obtain *finite graphs* as evaluation results, instead of *infinite trees*, because our goal is "graph" transformation.

In summary, we have to deal with these problems in order to apply the program-manipulation techniques studied in the programming language community to the graph transformation problem.

1.2 Contributions

In this paper, first, after a brief review of UnCAL (Section 3), we formalize a translation from UnCAL programs—which manipulate finite graphs—to functional programs that manipulate infinite trees (Section 4). The translation just follows the idea shown in Section 1.1. The purpose of Section 4 is to clarify the relationship between UnCAL programs and *usual* functional programs. This translation is useful also to reason about UnCAL programs as functional ones, and then is useful to import verification techniques (Section 2.1).

Second, to optimize UnCAL programs as functional ones, we give a semantics (Section 5) and a type system (Section 6) for the target language of the translation; a well-typed functional program under the type system can be executed as a *finite-graph* transformation under the semantics with *termination guarantee* (Section 7). We also show that the translated functional programs are well-typed, and that semantics in Section 4 and that in Section 5 "coincide". Thanks to the type system, users can freely optimize translated programs and finally run them as graph transformations, as long as the optimization keeps typability (Section 2.2). Note that our semantics itself is not new and nothing special; it is just the lazy semantics [34] with the black hole [2, 3, 34] and memoization. This helps us to implement the semantics easily, which runs faster than the existing implementation of UnCAL [23] (Section 2.3). Also, this enables us to bidirectionalize [29, 30] of UnCAL transformations (Section 2.4).

The main contributions of this paper are summarized as below:

- We formalize the transformation from UnCAL to functional ones to support reasoning of UnCAL programs as functional ones (Section 4).
- We give the semantics and the type system so that we can optimize the translated functional programs and execute them as graph transformations (Sections 5, 6 and 7).
- We show some applications of the proposed translation, semantics, and type system (Section 2).

2. Benefits

We start the paper with showing the benefits of our results, i.e., the translation from UnCAL to functional programs so that we can reason about UnCAL programs as functional ones, and the semantics and type system to support optimization and execution of UnCAL programs via functional ones. Also, our result enables us to import bidirectionalization techniques [29, 30, 41] studied for functional programs.

2.1 Verification

A verification problem of graph transformation is, given sets A and B and a transformation f, to check if $\forall a \in A. f(a) \in B$ holds or not. For XML transformations, these sets A and B are usually described in DTD, XML Schema, or RELAX NG. For model transformations seen in software engineering, they are often described in KM3 [26].

A few but interesting results are known for the verification problem on UnCAL. Buneman et al. [7] represent graph schemata (A and B above) again in graphs so that they can directly compute the image $f(A)$ by simply applying f to (a graph of) A. Inaba et al. [24] reduce the problem to the validity checking of monadic second-order logic (MSO) formulae when A and B are also given in MSO (fragments that respect bisimilarity), with some type annotations to a program f by users.

Our translation from UnCAL programs to functional ones enables us to access alternative methods, because the translation also reduces the verification problem for UnCAL programs to that for functional ones, which manipulates infinite trees instead of graphs. For example, thanks to our translation, we can use a verification method by Unno et al. [40] for graph transformations; it is originally designed for tree transformations written in (higher-order) functional programs where the trees can be infinite.

Although Inaba et al. [24]'s method is well tailored to UnCAL/UnQL and thus the benefits are rather small for the "current" UnCAL/UnQL, the advantage of our translation becomes clearer when we extend UnCAL/UnQL. For example, if we extend UnCAL/UnCAL to include higher-order functions to improve the programmability as in [22], then Inaba et al. [24]'s method becomes no longer applicable due to the higher-order constructs. In contract, the method by Unno et al. [40] is applicable for such extensions because it originally targets higher-order functional programs.

2.2 Optimization

Optimization is an important issue also in graph transformation. There have been a few approaches for optimization of UnCAL programs [8, 21]. The basic idea of these approaches is to elaborate the fact that UnCAL transformations respect bisimilarity and to rewrite **srec** as if it were defined as a recursive function on infinite trees as mentioned in Section 1.1. In addition to this basic idea, Hidaka et al. [21] also focus on manipulations of markers; for example, for $e\,@\,(\&x \triangleright e')$, their transformation statically computes the plugging-in operation by substituting $\&x$ in e by e', and sometimes replaces the expression to e statically or dynamically if e does not contain the output marker $\&x$.

The relationship between these UnCAL-specific techniques and usual optimization techniques for functional programs becomes clearer by our translation. Since our translation maps **srec** to fold on graphs as will be shown in Section 4, we can reinterpret the basic idea of their optimization as a special case of the classical fold-fusion [32]. Since the expression $e@(\&x \triangleright e')$ is converted to expression $(\lambda x.e)\,e'$ by our translation, we can regard Hidaka et al. [21]'s optimization as simplification by β-reduction. Both techniques are well understood in the programming language community.

In addition, our translation enables us to access heavier or lighter alternatives. To the translated programs from UnCAL, we can apply optimization methods safely, as long as the optimization preserves typability with respect to the type system in Section 6. For example, on the one hand, when the execution time will be more significant the compilation/optimization, we can use heavy but effective optimization methods such as supercompilation [38]. On the other hand, when the compilation time is as important as execution time, which is also typical in DB-querying, lighter-weight approaches such as short-cut fusion [17] and lightweight fusion [36] are preferable.

2.3 Implementation

Another benefit of our translation is that we can execute UnCAL programs as functional ones according to the lazy semantics in Section 5. Here, we report our experimental results, which show that UnCAL programs are executed significantly faster by our translation, for small graphs that can be loaded into a memory.

	$\lvert V\rvert$	$\lvert E\rvert$	Ours	GRoundTram	Speed Up
`Class2RDB`	55	73	**0.041**	0.16	3.9
`PIM2PSM`	46	58	**0.005**	0.032	6.4
`C2O_Sel`	25	45	**0.003**	0.014	4.7
`a2d_xc S30k`	30000	29999	**0.86**	3.7	4.3
`a2d_xc M200`	40000	80000	**3.7**	5.5	1.5
`a2d_xc C200`	201	40200	**1.5**	2.0	1.3

Table 1. Experimental results (running time is in CPU seconds).

We implemented our semantics in Section 5 as an embedded DSL on Haskell.[1] Although we cannot use GHC's lazy evaluation mechanism directly mainly due to the memoization we use (Section 5), we can explicitly represent memoized computation by using a monad. It is expected that the overhead of graph operations in a naive implementation, i.e., the overhead of directly handling sets of nodes and edges, or more precisely (V, E, I, O) quadruples as will be shown in Section 3, disappears in the implementation.

We measured execution time of a few transformations and compared the execution time with GRoundTram [20, 21, 23][2], an UnCAL implementation using OCaml. GRoundTram is implemented basically according to the semantics that will be shown in Section 3, with some optimizations [8, 21]. The experiments were held on MacOSX 10.8.3 over MacBook Air 11-inch with 1.4 GHz Intel Core 2 Duo CPU and 4 GB memory. We used GHC 7.6.3 (with LLVM 3.3) for Haskell and ocaml 4.00.1 for OCaml. The purpose of the experiments is to measure how the semantics in Section 5 is useful to implement UnCAL graph operations. Thus, we did not compile the tested programs and graphs; we used `runhaskell` instead while we compiled our embedded library providing the primitive graph operations. The examined programs were: `Class2RDB` is a benchmarking model transformation [5], `PIM2PSM` (from [21]) converts a platform independent model to a platform specific model, and `C2O_Sel` (from [21]) converts a customer-order database from a customer-oriented representation to a order-oriented representation with some extraction, and `a2d_xc` (from [20]) renames `a` to `d` and contracts `c`. The program codes of `Class2RDB`, `PIM2PSM` and `C2O_Sel` are mechanically generated; they are originally written in UnQL$^+$ [23] and converted to UnCAL. For `a2d_xc`, we used the three graphs as input: `S30k` is a 30000-long sequence of $\searrow\circ \xrightarrow{\mathtt{A}} \circ \xrightarrow{\mathtt{A}} \cdots \xrightarrow{\mathtt{A}} \circ$, `M200` is a lattice-like graph of 40000 nodes in which the i-th node connects to the $(i+1)$-th and $(i+200)$-th nodes modulo 40000 by `A`, and `C200` is a graph of 201 nodes in which every node connects to the other nodes by `A`.

Table 1 shows the experimental results. In all the examined cases, our implementation ran significantly faster than GRoundTram. The main source of the speed-up would be that there is no overhead of manipulating nodes and edges, which are stored in `Set`s in ocaml, in our implementation. Since GRoundTram already equips lazy-like evaluation for some special cases [21], the laziness of the translated program would not contribute to the speed-up so much. However, the speed-up is more significant in `Class2RDB`, `PIM2PSM` and `C2O_Sel`, which contain 114, 74 and 28 applications of **srec**s, respectively.

2.4 Bidirectionalization

A bidirectional transformation is a pair of a usual transformation of type $S \to V$ and a "backward" transformation of type $S \to V \to S$ that maps changes on the transformed data to the original data [15, 16, 20, 29, 30, 41]. A classic instance of bidirectional transformation is the "view updating problem" studied in the database community [4, 11], where a view is a result of a query (i.e., a transformation). Recently, bidirectional model transformation has been studied in the software engineering community to synchronize high-level model and low-level implementation [42, 43]. Bidirectional semantics of UnCAL [20] has been studied for this application because usually those models are represented by graphs.

Our translation enables us to access bidirectionalization methods [29, 30, 41] studied for functional programs. In [29, 30], if a function f has polymorphic type

$$\forall\alpha.\forall\mu.\mathit{PackM}\ c\ \alpha\ \mu \Rightarrow T\ \alpha \to \mu\ (T'\ \alpha)$$

where T and T' are type constructors for container-like datatypes, namely instances of $\mathit{Traversable}$, and $\mathit{PackM}\ c\,\alpha\,\mu$ is a type class with the methods below[3]

$$\begin{array}{l} \mathit{new} :: c \to \alpha \\ \mathit{liftO} :: \mathit{Eq}\ r \Rightarrow (c \to r) \to \alpha \to \mu\ r \\ \mathit{liftO2} :: \mathit{Eq}\ r \Rightarrow (c \to c \to r) \to \alpha \to \alpha \to \mu\ r \end{array}$$

then we can derive a backward transformation of type $T\ c \to T'\ c \to T\ c$ corresponding to $f_{c,I} :: T\ c \to I\ (T'\ c)$ where α and μ are instantiated by c and the identity monad I respectively.

It is rather straightforward to write an interpreter according to the semantics discussed in Section 5 that has the following type.

$$\mathit{eval} :: \mathit{PackM}\ \mathsf{L}\ \alpha\ \mu \Rightarrow \mathit{Exp}\ \to \mathit{Env}\ \alpha \to \mu\ (\mathit{Graph}\ \alpha)$$

Here, Exp is a type for translated expressions, and Env and Graph respectively are types for environments and graphs polymorphic in graph labels. The idea to write such an interpreter is that (1) instead of using label constant `a` directly we use new `a`, and (2) instead of direct label comparison $l = \mathtt{a}$ or $l = l'$, we use lifted comparisons $\mathit{liftO}\ (= \mathtt{a})\ l$ or $\mathit{liftO2}\ (=)\ l\ l'$.

There already exists a bidirectional semantics [20] of UnCAL. Although their method can handle insertions and deletions of edges in addition to label updates, it is very complex even for label updates while our approach is much simpler. Also, their method is not robust for language extension. For example, if we extend UnCAL/UnCAL to include higher-order functions, then Hidaka et al. [20]'s method becomes no longer applicable due to the higher-order constructs. In contrast, Matsuda and Wang [29, 30]'s method is still applicable because its only requirement is the polymorphic type above and thus it allows us to use higher-order functions in a forward transformation.

3. Brief Overview of UnCAL

In this section, we briefly overview UnCAL [8], a first-order functional programming language that manipulates graphs. UnCAL is an internal language of UnQL [8] and its extension UnQL$^+$ [23], in which users can write query like SQL; for example, the query that extracts a person named `Bob` shown in Section 1.1 can be written as follows.

```
select $p where
  $p in $db, {name: $n, friend: $f} in $p,
  {$nameL: {}} in $n, $nameL = Bob
```

Once a query is written in UnQL/UnQL$^+$, it is converted to an UnCAL program and then is executed. This means, our translation is also beneficial to UnQL/UnQL$^+$. It is remarkable that, recently, UnCAL is applied to bidirectional model-driven software development [20, 23, 43], where a structure of a software is modeled as a graph in different levels of abstractions and their relationships are described by graph transformations.

[1] The implementation is available from `https://bitbucket.org/kztk/funcal`

[2] `http://www.biglab.org/download.html`

[3] In the original paper, PackM has a method to lift n-ary functions instead of those special for unary or binary ones.

3.1 Graphs in UnCAL

UnCAL deals with *multi*-rooted, directed, and edge-labeled graphs with no order on outgoing edges. The characteristic points of the UnCAL graphs are: (1) the UnCAL graphs can have *markers* that indicate roots and holes, (2) the UnCAL graphs can have ε-edges that have similar behaviors to ε-transitions in automata, and (3) the equivalence of the UnCAL graphs are defined by bisimulation.

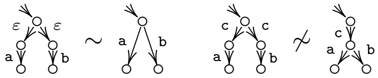
As mentioned in Section 1.1, markers are used in the two ways: *input* and *output*. Input markers are names for multiple roots, and output markers are names for holes. Nodes may be marked with input and output markers (*input nodes* and *output nodes*), and they can be connected to produce other graphs; e.g., one can construct cycles by connecting nodes with input markers to that of output markers of the same names in the same graph.

UnCAL graphs can contain ε-edges representing "short-cuts", similarly to the ε-transitions in automata. For example, if a node v is connected to a node u by an ε-edge, it means that the edges of u are also edges of v (the converse is not necessarily true because v can have other edges than this ε-edge). UnCAL uses ε-edge to delay some graph operations for efficiency [8].

We define the UnCAL graphs formally. Let $\mathcal{M}$ be a set of markers and L be a certain set of labels. An *UnCAL graph* G is a quadruple (V, E, I, O), where V is a set of nodes, $E \subseteq V \times (\mathsf{L} \cup \{\varepsilon\}) \times V$ is a set of edges, $I \subseteq \mathcal{M} \times V$ is a set of pairs of input markers and the corresponding input nodes, and $O \subseteq V \times \mathcal{M}$ is a set of pairs of output nodes and associated output markers. In addition, we require that, for each marker $\&x \in \mathcal{M}$, there is at most one node v such that $(\&x, v) \in I$. In other words, I is a partial function from markers to nodes and is sometimes denoted as such. For a singly-rooted graph, the default marker $\&$ is often used to indicate the root. We call the markers in the sets $\{\&x \mid (\&x, \_) \in I\}$ and $\{\&x \mid (\_, \&x) \in O\}$ *input* and *output* markers, respectively. Throughout this paper, we fix the (denumerable) set of labels L.

The equivalence between UnCAL graphs is defined by bisimulation extended with ε-edges. Intuitively, two UnCAL graphs are equivalent if the infinite trees obtained by unfolding sharings and cycles are identical, after short-cutting all the ε-edges. Let us define the bisimilarity between graphs formally. We write $v \to^{l} u$ if there is an edge $(v, l, u) \in E$ between nodes $v, u \in V$ in a graph $G = (V, E, I, O)$, and write $\overset{*}{\to}{}^{\varepsilon}$ for the reflexive transitive closure of $\to^{\varepsilon}$. A *bisimulation* $\mathcal{X}$ between a graph $G_1 = (V_1, E_1, I_1, O_1)$ and $G_2 = (V_2, E_2, I_2, O_2)$ is a relation satisfying the following conditions: (1) if $(v_1, v_2) \in \mathcal{X}$, for any path satisfying $v_1 \overset{*}{\to}{}^{\varepsilon} w_1 \to^{\mathtt{a}} u_1$ there is a path satisfying $v_2 \overset{*}{\to}{}^{\varepsilon} w_2 \to^{\mathtt{a}} u_2$ and $(u_1, u_2) \in \mathcal{X}$, and for any path u_2 satisfying $v_2 \overset{*}{\to}{}^{\varepsilon} w_2 \to^{\mathtt{a}} u_2$ there is a path satisfying $v_1 \overset{*}{\to}{}^{\varepsilon} w_1 \to^{\mathtt{a}} u_1$ and $(u_1, u_2) \in \mathcal{X}$; (2) if $(v_1, v_2) \in \mathcal{X}$, for any path $v_1 \overset{*}{\to}{}^{\varepsilon} u_1$ such that $(u_1, \&x) \in O$, there is a path $v_2 \overset{*}{\to}{}^{\varepsilon} u_2$ such that $(u_2, \&x) \in O$, and conversely, for any path $v_2 \overset{*}{\to}{}^{\varepsilon} u_2$ such that $(u_2, \&x) \in O$, there is a path $v_1 \overset{*}{\to}{}^{\varepsilon} u_1$ such that $(u_1, \&x) \in O$; and (3) $\mathsf{dom}(I_1) = \mathsf{dom}(I_2)$ and $(I_1(\&x), I_2(\&x)) \in \mathcal{X}$ for any $\&x \in \mathsf{dom}(I_1) = \mathsf{dom}(I_2)$. Two graphs G_1 and G_2 are called *bisimilar*, denoted by $G_1 \sim G_2$, if there is a bisimulation between G_1 and G_2.

Note that the graph bisimulation is different from weak bisimulation [33] or the equivalence of the languages of automata. The following examples illustrate the difference.



The bisimilarity of the first two examples shows the difference from weak bisimulation; recall that ε-edges represent shortcuts. The non-bisimilarity of the last two examples shows the difference from the equivalence of the trace sets, or the equivalence of automata.

$$
\begin{array}{rcll}
g & ::= & \{\} \mid \{l : g\} \mid g_1 \cup g_2 & \text{(graph constructors)} \\
 & & \mid \&x \rhd g \mid \&y \mid () \mid g_1 \oplus g_2 & \\
 & & \mid g_1 \mathbin{@} g_2 \mid \mathbf{cycle}(g) & \\
 & \mid & \mathbf{srec}(\lambda(l, t).g_1)(g_2) & \text{(structural recursion)} \\
 & \mid & t & \text{(graph variable)} \\
l & ::= & l \mid \mathtt{a} &
\end{array}
$$

Figure 3. The syntax of the positive subset of UnCAL

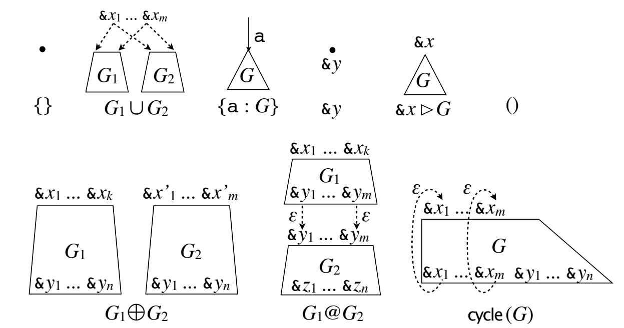


Figure 4. Graph Constructors

The bisimilarity-based semantics of the UnCAL graphs supports usual equational reasoning of recursive functions on graphs [8, 21]. The purpose of this paper is to further investigate this direction and show a closer relationship between UnCAL programs and functional ones.

3.2 Syntax and Semantics

Figure 3 shows the positive [8] subset of UnCAL that we mainly target in this paper. The subset of UnCAL consists of the nine graph constructors and **srec** for a *structural recursion*. Compared with full UnCAL [8], the subset does not contain **if**-expressions and the **isEmpty** operator that checks if a graph has at least one non-ε-edge accessible from the roots or not. The former restriction is just for simplicity; we can extend our discussions straightforwardly to **if**-expressions.

In contrast, we have to be more careful with **isEmpty**. As we will discuss in the end of Section 4, there is no computable counterpart of **isEmpty** in *general* functional programs, and it must be converted to an oracle. However, unlike the discussion in Section 4, the discussions in Sections 5, 6 and 7 can be easily extended to **isEmpty** because **isEmpty** becomes computable for the class of functional programs restricted by the type system in Section 6.

3.2.1 Graph Constructors

UnCAL has the nine graph constructors, $\{\}$, $\{\_ : \_\}$, $\cup$, $\&x \rhd \_$, $\&y$, $()$, $\oplus$, $@$, and **cycle**. Some of them are already mentioned in Section 1.1. A record notation shown in Section 1.1 such as $\{\mathtt{name} : \mathtt{Alice}, \mathtt{email} : \mathtt{alice}\}$ is a syntax sugar for $\{\mathtt{name} : \{\mathtt{Alice} : \{\}\}\} \cup \{\mathtt{email} : \{\mathtt{alice} : \{\}\}\}$. Figure 4 illustrates their intuitive behaviors. In what follows, we introduce the formal definitions of these nine constructors each by each.

Singleton Graph The expression $\{\}$ constructs a (single) root-only graph, of which semantics $[\![\{\}]\!]$ is defined by:

$$[\![\{\}]\!] = (\{v\}, \emptyset, \{\& \mapsto v\}, \emptyset)$$

Here v is a fresh node.

Edge Extension The expression $\{\mathtt{a} : g\}$ constructs a graph by adding an edge with label $\mathtt{a}$ pointing to the root of the graph

$[\![g]\!]$; formally, its semantics is defined by:

$$[\![\{\mathtt{a}:g\}]\!] = (V \cup \{u\}, E \cup \{(u,\mathtt{a},v)\}, \{\& \mapsto u\}, O) \\ \text{where } (V,E,\{\& \mapsto v\},O) = [\![g]\!]$$

Here, u is a fresh node.

Edge-set Union The expression $g_1 \cup g_2$ adds two ε-edges from the new root to the roots of G_1 and G_2, where G_1 and G_2 are evaluation results of g_1 and g_2, respectively. Formally, its semantics is defined by:[4]

$$\begin{aligned}[\![g_1 \cup g_2]\!] = (V, E, \{\& \mapsto v\}, O_1 \cup O_2) \\ \text{where } & (V_1,E_1,I_1,O_1) = [\![g_1]\!] \\ & (V_2,E_2,I_2,O_2) = [\![g_2]\!] \\ & V = V_1 \cup V_2 \cup \{v\} \\ & E = E_1 \cup E_2 \cup \{(v,\varepsilon,I_1(\&)),(v,\varepsilon,I_2(\&))\}\end{aligned}$$

Here, v is a fresh node, and V_1 and V_2 are assumed to be disjoint.

Named Hole The expression $\&y$ constructs a graph with a single node marked with an output marker $\&y$, of which semantics is defined by:

$$[\![\&y]\!] = (\{v\}, \emptyset, \{\& \mapsto v\}, \{(v,\&y)\})$$

Here v is a fresh node.

Naming Root The expression $\&x \rhd g$ names the root of $[\![g]\!]$ by $\&x$, of which semantics is defined by:

$$[\![\&x \rhd g]\!] = (V, E, \{\&x \mapsto v\}, O) \\ \text{where } (V,E,\{\& \mapsto v\},O) = [\![g]\!]$$

Unlike the original definition [8], we restrict that the input markers of g must be the singleton $\{\&\}$ for simplicity.

Root-set Union The expression $g_1 \oplus g_2$ combines two graphs $[\![g_1]\!]$ and $[\![g_2]\!]$ with different sets of input markers, of which semantics is defined by:

$$\begin{aligned}[\![g_1 \oplus g_2]\!] = (V_1 \cup V_2, E_1 \cup E_2, I_1 \cup I_2, O_1 \cup O_2) \\ \text{where } & (V_1,E_1,I_1,O_1) = [\![g_1]\!] \\ & (V_2,E_2,I_2,O_2) = [\![g_2]\!]\end{aligned}$$

Here, we assume that V_1 and V_2 are disjoint, and require that $\mathsf{dom}(I_1)$ and $\mathsf{dom}(I_2)$ are disjoint.

Empty Graph The expression $()$ represents a graph with no nodes or edges, i.e.,

$$[\![()]\!] = (\emptyset,\emptyset,\emptyset,\emptyset)$$

Plugging In The expression $g_1 \mathbin{@} g_2$ replaces holes in $[\![g_1]\!]$ with roots of $[\![g_2]\!]$ that share the same names, of which semantics is defined by:

$$\begin{aligned}[\![g_1 \mathbin{@} g_2]\!] = (V, E, I_1, O_2) \\ \text{where } & (V_1,E_1,I_1,O_1) = [\![g_1]\!] \\ & (V_2,E_2,I_2,O_2) = [\![g_2]\!] \\ & V = V_1 \cup V_2 \\ & E = E_1 \cup E_2 \cup \{(v,\varepsilon,I_2(\&x)) \mid (v,\&x) \in O_1\}\end{aligned}$$

Here, we assume that V_1 and V_2 are disjoint, and require $\mathsf{ran}(O_1) \subseteq \mathsf{dom}(I_1)$.

Cycle The expression $\mathbf{cycle}(g)$ constructs cycles by replacing holes with roots in $[\![g]\!]$ that share the same names, of which semantics is defined by:

$$\begin{aligned}[\![\mathbf{cycle}\, g]\!] = (V, E', I, \{(v,\&x) \in O \mid \&x \notin \mathsf{dom}(I)\}) \\ \text{where } & (V,E,I,O) = [\![g]\!] \\ & E' = E \cup \{(v,\varepsilon,I(\&x)) \mid (v,\&x) \in O\}\end{aligned}$$

Some examples have been shown already in Section 1. The following is an alternative way to define the cyclic graph in Section 1.

$$\begin{aligned}\&a \mathbin{@} \mathbf{cycle}((&\&a \rhd (\{\mathtt{name}:\{\mathtt{Alice}:\{\}\}\} \cup \{\mathtt{friend}:\&b\})) \\ &\oplus (\&b \rhd (\{\mathtt{name}:\{\mathtt{Bob}:\{\}\}\} \cup \{\mathtt{friend}:\&a\})))\end{aligned}$$

Structural Recursion The expression $\mathbf{srec}(\lambda(l,t).g)(\_)$ represents a *structural recursion* in the sense that a function $f(x) = \mathbf{srec}(\lambda(l,t).g)(x)$ satisfies the following laws [8].

$$f(\{\}) = \{\} \tag{SR1}$$
$$f(\{\mathtt{a}:G\}) = g[\mathtt{a}/l, G/t] \mathbin{@} f(G) \tag{SR2}$$
$$f(G_1 \cup G_2) = f(G_1) \cup f(G_2) \tag{SR3}$$

Thanks to $\mathbf{srec}$, UnCAL can express many graph transformations in an efficient way, with guarantee of termination [8, 23].

Formally, its semantics is defined by:

$$\begin{aligned}[\![\mathbf{srec}(\lambda(l,t).g)(g')]\!] = (V' \cup \textstyle\bigcup_{\zeta \in E} V_\zeta, E' \cup \bigcup_{\zeta \in E} E_\zeta, \\ \{\&x \mapsto u_{v_0,\&x} \mid \&x \in Z\}, \emptyset)\end{aligned}$$

$$\begin{aligned}\text{where } & \\ & (V,E,\{\& \mapsto v_0\},\emptyset) = [\![g']\!] \\ & (V_\zeta,E_\zeta,I_\zeta,O_\zeta) = [\![g[\mathtt{a}/l,(V,E,\{\& \mapsto v\},\emptyset)/t]]\!] \\ & \qquad (\zeta = (\_,\mathtt{a},v)) \\ & V' = \{u_{v,\&x} \mid v \in V, \&x \in Z\} \\ & E' = \{(v',\varepsilon,u_{v,\&x}) \mid \exists \mathtt{a},u.\ (v',\&x) \in O_{(u,\mathtt{a},v)}\} \\ & \quad \cup \{(u_{v,\&x},\varepsilon,v') \mid \exists \mathtt{a},u.\ I_{(v,\mathtt{a},u)}(\&x) = v'\}\end{aligned}$$

Here, V' are fresh nodes, and $Z = \mathsf{dom}(I_\zeta)$ and $\mathsf{ran}(O_\zeta) \subseteq Z$ for each $\zeta \in E$. We assume that $\mathsf{dom}(I_\zeta) = \mathsf{dom}(I_{\zeta'})$ for all $\zeta,\zeta' \in E$, and V_ζ and $V_{\zeta'}$ are disjoint for different $\zeta,\zeta' \in E$. Intuitively, the semantics computes $g[\mathtt{a}/l, G/t]$ for each edge in $[\![g']\!]$, and connects them by ε-edges which corresponds to (SR2) and (SR3). Unlike the original definition [8], we require the graph $[\![g']\!]$ to have only one root named $\&$ and no holes. The former restriction is just for simplicity. For typed UnCAL [8], we can convert UnCAL programs to ones that satisfy the condition. In contrast, the latter restriction reduces the expressive power to some extent. However, UnCAL programs that violate the latter restriction are rare in practice. For example, UnCAL programs obtained from the surface languages UnQL and UnQL$^+$ satisfy the restriction.

3.3 Types

One would notice that there are some conditions on markers to perform some graph constructions such as $\cup$. To guarantee these conditions, UnCAL has a type system concerning markers [6, 21], in which a graph type is of the form DB^X_Y. A type DB^X_Y represents a set of the graphs whose input markers are *exactly* X, and whose output markers are *contained in* Y. For example, expression $\&y$ can have type DB^X_Y where $X = \{\&\}$ and $Y \supseteq \{\&y\}$. Recall that $\&$ is the marker to refer roots obtained from $\{\}$, $\{\_:\_\}$ and $\&y$. We shall omit the typing rules, because it is straightforward and one can extract the typing rules from the conversion rules from UnCAL to functional programs, which will be shown in the next section.

4. UnCAL Programs to Functional Programs

In this section, we formalize a translation from UnCAL programs to functional ones, whose idea has been roughly mentioned in Section 1.1. The translation enables us to reason about UnCAL programs as functional ones and thus to import verification techniques for functional programs to the graph transformation problem, as discussed in Section 2.

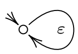
[4] In the definition, we restrict that g_i $(i = 1, 2)$ has only one root while the original definition allows g_1 and g_2 to have multiple roots. This is handy when we convert UnCAL to functional programs. This restriction does not lose the expressive power; for g_1 and g_2 with input markers $\&x_1,\ldots,\&x_n$, the original $g_1 \cup g_2$ can be rewritten as $(\&x_1 \rhd ((\&x_1 @ g_1) \cup (\&x_1 @ g_2))) \oplus \ldots \oplus (\&x_n \rhd ((\&x_n @ g_1) \cup (\&x_n @ g_2)))$, in which $\cup$ satisfies this restriction.

$e ::= x \mid \lambda x.e \mid e_1\ e_2$	(λ-terms)
$\mid\ \pi_i^n\ e \mid (e_1, \ldots, e_n)$	(projections and tuples)
$\mid\ \mathsf{a}$	(labels)
$\mid\ e_1 : e_2 \mid e_1 \cup e_2 \mid \bullet$	(graph constructors)
$\mid\ \mathsf{fix}_{\mathbf{G}}\ e$	(first-order fixed-point operator)
$\mid\ \mathsf{fold}_n e$	(structural recursion for graphs)

Figure 5. The syntax of FUnCAL, the target language of the translation.

As mentioned in Section 1.1, we convert an UnCAL program that manipulates graphs to a functional one that manipulates infinite trees, by emulating the UnCAL specific features, *markers* and their manipulation, by standard notions in functional programming languages. Specifically, we emulate input markers—names for roots—by tuples, and output markers—names for holes—by λ-abstractions.

It is true that a translated functional program can be nonterminating; for example, an UnCAL expression $\mathbf{cycle}(\&)$ satisfying

$$[\![\mathbf{cycle}(\&)]\!] =$$  $$\sim [\![\{\}]\!]$$

is converted to $\mathsf{fix}_{\mathbf{G}}\ (\lambda x.x)$ that diverges. However, this is rather natural and not problematic in the bisimulation-based reasoning, which will be shown in Section 4.2. Recall that, in process calculi, (strong or weak) bisimilarity cannot distinguish a terminating process from a nonterminating process if each of them does not interact other processes. How to execute the translated programs as graph transformations will be discussed in Sections 5, 6 and 7.

Recall that g of $\mathbf{srec}(\ldots)(g)$ is restricted not to contain output markers. This is a key to regarding output markers as holes. For example, in the original UnCAL without the restriction, $\mathbf{srec}(\lambda(l,t).g')(\&y) = \&y$ holds for g' with type $\mathsf{DB}^{\{\&\}}_{\{\&\}}$. This behavior is different from that of holes; if the output markers are holes, a graph substituted to a hole must be traversed by the $\mathbf{srec}$.

4.1 Translation

The syntax of the target language of our translation is given in Figure 5. We call the language FUnCAL. The target language contains λ-expressions, tuples (where π_i^n is the projection of the ith element from an n tuple), (infinite) tree constructors, and the (first-order) fixed-point operator $\mathsf{fix}_{\mathbf{G}}$, and structural recursions $\mathsf{fold}_n\ f$. Tree constructors consist of a leaf $\bullet$, edge extension $(:)$, and branch construction $\cup$. For simplicity, we shall write $a : b$ for $\{a : b\}$ henceforth. We use $\mathsf{fix}_{\mathbf{G}}$ instead of $\mathbf{letrec}$ that appeared in Section 1.1, since it is handy to discuss reductions. The structural recursion $\mathsf{fold}_n\ f$ is "fold" for the tree constructors defined recursively as:

$$\mathsf{fold}_n\ f\ \bullet = (\bullet, \ldots, \bullet) \qquad \text{(Fold1)}$$
$$\mathsf{fold}_n\ f\ (x : y) = f\ x\ (\mathsf{fold}_n\ f\ y) \qquad \text{(Fold2)}$$
$$\mathsf{fold}_n\ f\ (x \cup y) = (\mathsf{fold}_n\ f\ x) \cup (\mathsf{fold}_n\ f\ y) \qquad \text{(Fold3)}$$

Note that fold_n returns an n-tuple of trees rather than a tree; in the right-hand side of (Fold3), we overload $\cup$ to tuples as $(x_1, \ldots, x_n) \cup (y_1, \ldots, y_n) = (x_1 \cup y_1, \ldots, x_n \cup y_n)$. Unlike general "fold", the operations for $\bullet$ and $\cup$ are fixed in the definition.

As we have mentioned earlier, in our translation, we emulate input markers by tuples and output markers by λ-abstractions. Thus, an UnCAL expression $g :: \mathsf{DB}^X_Y$ is translated to

$$e :: \mathsf{G}^{|Y|} \to \mathsf{G}^{|X|}$$

where G is the (coinductive) datatype defined by:

$$\mathbf{data}\ \mathsf{G} = \bullet \mid \mathsf{L} : \mathsf{G} \mid \mathsf{G} \cup \mathsf{G}$$

In this section, we assume that FUnCAL has the standard simple type system with the datatype G and the label type L; we later refine it to guarantee termination (Section 6). For example, an expression $\lambda f.\mathsf{fold}_n f$ has type $(\mathsf{L} \to \mathsf{G}^n \to \mathsf{G}^n) \to \mathsf{G} \to \mathsf{G}^n$.

We introduce several notations. We sometimes shall write π_i instead of π_i^n if n is clear from the context. We assume that markers are totally ordered, and write $\overline{X}$ for a tuple $(x_1, \ldots, x_n)$ where $\{\&x_1, \ldots, \&x_n\} = X$ and $\&x_i < \&x_j$ $(i < j)$. Sometimes, we use a syntax sugar $\lambda\overline{X}.e$ for $\lambda t.e[(\pi_i t)/x_i]_{1 \le i \le n}$ where $(x_1, \ldots, x_n) = \overline{X}$. For example, assuming $x_1 < x_2$, we write $\lambda(x_1, x_2).x_1$ for $\lambda t.\pi_1 t$.

Our translation is defined according to the typing derivation of UnCAL. A translation judgment $\Gamma \vdash g :: \mathsf{DB}^X_Y \rightsquigarrow e$ reads that an expression g of type DB^X_Y under a typing environment Γ in UnCAL is converted to an expression e, where the type of g and types in Γ are converted from $\mathsf{DB}^{X'}_{Y'}$ to $\mathsf{G}^{|Y'|} \to \mathsf{G}^{|X'|}$. Figure 6 shows the translation rules for the UnCAL graph constructors. If we ignore the $\rightsquigarrow e$ part of $\Gamma \vdash g :: \mathsf{DB}^X_Y \rightsquigarrow e$, the judgment and rules coincide to the typing judgment and rules of UnCAL [6, 21].

The conversion rule for $\mathbf{srec}$ is a bit involved and thus is written separately as follows.

$$\frac{\Gamma, t :: \mathsf{DB}^{\{\&\}}_{\emptyset} \vdash g_1 :: \mathsf{DB}^Z_Z \rightsquigarrow e_1 \quad \Gamma \vdash g_2 :: \mathsf{DB}^{\{\&\}}_{\emptyset} \rightsquigarrow e_2}{\Gamma \vdash \mathbf{srec}(\lambda(l,t).g_1)(g_2) :: \mathsf{DB}^Z_{\emptyset} \rightsquigarrow \lambda y.\mathsf{para}_{|Z|}\ (\lambda l.\lambda t'.(\lambda t.e_1)\ (\lambda().t'))\ (e_2\ ())}\ \text{C-REC}$$

Here, $\mathsf{para}_n e$, representing a "paramorphism"[5] [31], where an expression $\lambda f.\mathsf{para}_n f$ has type $(\mathsf{L} \to \mathsf{G} \to \mathsf{G}^n \to \mathsf{G}^n) \to \mathsf{G} \to \mathsf{G}^n$, is a syntax sugar defined by:

$$\mathsf{para}_n e\ y \equiv p\left(\mathsf{fold}_{n+1}\left(\lambda z.\lambda x.q\ \left(e\ z\ (p'\,x)\ (p\ x)\right)\ \left(z : p'\,x\right)\right)\ y\right)$$

where $p :: \mathsf{G}^{n+1} \to \mathsf{G}^n$, $p' :: \mathsf{G}^n \to \mathsf{G}$ and $q :: \mathsf{G}^n \to \mathsf{G} \to \mathsf{G}^{n+1}$ are functions to rearrange tuples defined as $p\,x = (\pi_1 x, \ldots, \pi_n x)$, $p'\ x = \pi_{n+1} x$, and $q\ x\ y = (\pi_1 x, \ldots, \pi_n x, y)$. This definition of para_n is similar to how a paramorphism is represented by a catamorphism (fold) via tupling [31]. The rule C-REC becomes a bit complicated due to explicit conversion between G and $() \to \mathsf{G}$. Since the argument of para_n must be of type G instead of $() \to \mathsf{G}$, we apply $()$ to e_2. In addition, since the conversion assumes that t in e_1 has type $() \to \mathsf{G}$, we construct such a function by $\lambda().t'$.

For example, $\mathbf{cycle}(\{\mathsf{a} : \&\})$ of type $\mathsf{DB}^{\{\&\}}_{\emptyset}$ is converted to $\lambda y.\,\mathsf{fix}_{\mathbf{G}}(\lambda x.\mathsf{a} : x)$ of type $() \to \mathsf{G}$ after some simplification based on the standard β and η conversions. An UnCAL expression $\mathbf{srec}(\lambda(l,g).\&)(\mathbf{cycle}(\{\mathsf{b} : \&\}))$ is converted to $\lambda y.\mathsf{fold}_1(\lambda l.\lambda r.r)(\mathsf{fix}_{\mathbf{G}}(\lambda x.\mathsf{b} : x$ after some simplification.

It is not difficult to show the translated programs are well-typed.

4.2 Correctness

Although the translation is rather simple, some extra effort is required to state its correctness; we have to be careful with the following difference between UnCAL and FUnCAL: An UnCAL graph of type DB^X_Y, which can contain output markers in Y, is translated to a tree-to-tree *function* $\mathsf{G}^{|Y|} \to \mathsf{G}^{|X|}$ in FUnCAL rather than an expression that generates a (tuple of) tree. To leap the gap, we first define a relation between output-marker-free UnCAL graphs and FUnCAL tree expressions, and then we extend the relation to one that between general UnCAL graphs and FUnCAL functions.

First, we define a graph obtained from an expression as a labeled transition system [33].

Definition 1. A *reduction graph* $G_{e_0,X}$ of a (possibly-open) FUnCAL expression e_0 of type G^n and markers $X = \{\&x_1, \ldots, \&x_n\}$

[5] Precisely, paramorphism is a notion for inductive datatypes. We borrow the name just because the computation patterns are similar.

$$\frac{}{\Gamma \vdash \{\} :: \mathsf{DB}_Y^{\{\&\}} \rightsquigarrow \bullet}\text{C-SINGLE}$$

$$\frac{\Gamma \vdash g :: \mathsf{DB}_Y^{\{\&\}} \rightsquigarrow e}{\Gamma \vdash \{l : g\} :: \mathsf{DB}_Y^{\{\&\}} \rightsquigarrow \lambda y.l : (e\ y)}\text{C-EDGE}$$

$$\frac{\{\Gamma \vdash g_i :: \mathsf{DB}_Y^{\&} \rightsquigarrow e_i\}_{i=1,2}}{\Gamma \vdash g_1 \cup g_2 :: \mathsf{DB}_Y^{\&} \rightsquigarrow \lambda y.(e_1\ y) \cup (e_2\ y)}\text{C-UNI}$$

$$\frac{\Gamma \vdash g :: \mathsf{DB}_Y^{\{\&\}} \rightsquigarrow e}{\Gamma \vdash \&x \triangleright g :: \mathsf{DB}_Y^{\{\&x\}} \rightsquigarrow e}\text{C-ROOT}$$

$$\frac{\&y = \&y_i \text{ of } (\&y_1, \ldots, \&y_n) = \overline{Y}}{\Gamma \vdash \&y :: \mathsf{DB}_Y^{\{\&\}} \rightsquigarrow \lambda z.\pi_i z}\text{C-HOLE}$$

$$\frac{}{\Gamma \vdash () :: \mathsf{DB}_Y^{\emptyset} \rightsquigarrow \lambda y.()}\text{C-EMP}$$

$$\frac{\{\Gamma \vdash g_i :: \mathsf{DB}_Y^{X_i} \rightsquigarrow e_i\}_{i=1,2} \quad p \equiv \lambda\overline{X_1}.\lambda\overline{X_2}.\overline{X_1 \cup X_2}}{\Gamma \vdash g_1 \oplus g_2 :: \mathsf{DB}_Y^{X_1 \uplus X_2} \rightsquigarrow \lambda y.p\ (e_1\ y)\ (e_2\ y)}\text{C-RU}$$

$$\frac{\Gamma \vdash g_1 :: \mathsf{DB}_Y^X \rightsquigarrow e_1 \quad \Gamma \vdash g_2 :: \mathsf{DB}_Z^Y \rightsquigarrow e_2}{\Gamma \vdash g_1 \mathbin{@} g_2 :: \mathsf{DB}_Z^X \rightsquigarrow \lambda y.e_1\ (e_2\ y)}\text{C-SUBST}$$

$$\frac{\Gamma \vdash g :: \mathsf{DB}_{X \uplus Y}^X \rightsquigarrow e}{\Gamma \vdash \mathbf{cycle}(g) :: \mathsf{DB}_Y^X \rightsquigarrow \lambda\overline{Y}.\,\mathsf{fix}_{\mathbf{G}}(\overline{X}.e\ \overline{X \cup Y})}\text{C-CYC}$$

$$\frac{}{\Gamma \vdash y :: \Gamma(y) \rightsquigarrow y}\text{C-VAR}$$

$$\frac{\Gamma \vdash g :: \mathsf{DB}_Y^X \rightsquigarrow e \quad Y \subseteq Y'}{\Gamma \vdash g :: \mathsf{DB}_{Y'}^X \rightsquigarrow \lambda\overline{Y'}.e\ \overline{Y}}\text{C-SUB}$$

Figure 6. Conversion rules of UnCAL graph-constructors.

with $\&x_i < \&x_j$ $(i < j)$ is an (possibly-infinite) UnCAL graph $(V, E, I, \emptyset)$ where V is the set of FUnCAL expressions, $E = \{(e, \mathtt{a}, e') \mid e \Rightarrow^* \mathtt{a} : e'\}$, and $I = \{\&x_1 \mapsto \pi_1 e_0, \ldots, \&x_n \mapsto \pi_n e_0\}$. Here, the relation $(\Rightarrow)$ is defined by:

$$e_1 \cup e_2 \Rightarrow e_1 \qquad e_1 \cup e_2 \Rightarrow e_2 \qquad e \Rightarrow e' \text{ if } e \to e'$$

where, $\to$ is the call-by-name reduction. □

Note that, in each reduction by $\Rightarrow$, only $(\cup)$ occurring at the top is interpreted as nondeterministic choice. We write G_e for $G_{e,X}$ if X is clear from the context or not relevant in the context. We abuse the notation to write $G \sim e$ and $e \sim e'$ for $G \sim G_e$ and $G_e \sim G_{e'}$, respectively. Note that G such that $G \sim e$ for some e must not have output markers. By definition, if e and e' are equivalent as infinite constructor trees (i.e., $\cup$ is frozen), then $e \sim e'$.

Then, we define a correspondence $(\approx)$ between an UnCAL graph $G :: \mathsf{DB}_Y^X$ and an expression $e :: \mathsf{G}^{|Y|} \to \mathsf{G}^{|X|}$ by: $G \approx e$ iff $(G \mathbin{@} (G_1 \oplus \ldots \oplus G_{|Y|})) \sim (e\ (e_1, \ldots, e_{|Y|}))$ for any G_j, e_j $(1 \leq j \leq |Y|)$ such that $G_j :: \mathsf{DB}_\emptyset^{\&}$ and $G_j \sim e_j$.

Now, we have the following theorem.

Theorem 1 (Correctness)**.** *If* $\vdash g :: \mathsf{DB}_Y^X \rightsquigarrow e$*,* $[\![g]\!] \approx e$ *holds.*

Proof. See Appendix A. □

4.3 Translation of isEmpty

The full-set of UnCAL contains **isEmpty** as we have mentioned before. Although many transformation can be described without **isEmpty** [8] as those obtained from UnQL, there are still useful transformations that require **isEmpty**; for example, some UnCAL programs converted from UnQL$^+$, such as `Class2RDB` in Section 2.3, contain **isEmpty** [23].

Since a graph is translated to an infinite tree, it is natural that **isEmpty** is translated to the productivity test that checks whether e satisfies $e \Rightarrow^* \mathtt{a} : e'$ for some $\mathtt{a}$ and e', which is generally undecidable. In other words, **isEmpty** is translated to an oracle instead of a computable function according to our translation. This is the reason why we consider the positive subset of UnCAL. For the positive subset of UnCAL, we can reason about the UnCAL programs through translated functional programs. For the full-set of UnCAL, additional reasoning effort is needed to handle the productivity-test oracle, although special treatment of markers are not necessary in reasoning of the translated programs.

However, **isEmpty** does not pose any problems when we execute UnCAL programs as functional ones (Sections 5, 6, and 7). Roughly speaking, the type system in Section 6 ensures that the productivity test is *decidable* for the well-typed programs.

5. Evaluating Functional Programs as Finite-Graph Transformations

The translation in Section 4 enables us to reason about UnCAL programs as functional ones; for example, we can apply verification techniques studied for functional programs to UnCAL ones (Section 2.1). However, the translated programs are not graph transformations; i.e., according to the usual semantics, they may result in infinite trees rather than finite graphs.

In this section, we give a semantics based on Nakata and Hasegawa [34]'s lazy semantics, which extends Launchbury [28]'s natural semantics with the *black hole* [2, 3, 34] ("apparent undefinedness"), so that a FUnCAL program runs as a finite-graph transformation. This section focuses on the formal description of the semantics. Formal discussions about termination will be postponed to Sections 6 and 7.

The basic idea is to exploit a pointer-structure in a heap under the lazy evaluation. For example, the heap obtained after evaluation of $\mathsf{fix}_{\mathbf{G}}\ (\lambda x.\mathtt{a} : x)$ in the usual lazy evaluation is cyclic and has a similar structure to a corresponding UnCAL graph $[\![\mathbf{cycle}(\{\mathtt{a} : \&\})]\!]$. However, an extra effort is required to handle $\mathsf{fix}_{\mathbf{G}}(\lambda x.x)$ and $\mathsf{fold}(\lambda a.\lambda r.r)(e)$ for example, which are nonterminating in usual semantics.

The lazy semantics with the black hole [34] plays an important role to resolve the problem. Since $\mathsf{fix}_{\mathbf{G}}(\lambda x.x)$ evaluates to the black hole without running infinitely in the lazy semantics, we identify the black hole with a singleton graph, and obtain a singleton graph as the evaluation result of $\mathsf{fix}_{\mathbf{G}}(\lambda x.x)$. Still, the semantics is not sufficient for terminating evaluation of recursions such as $\mathsf{fold}(\lambda a.\lambda r.r)(e)$. To make the evaluation of recursions terminating, we adopt *memoization*. Roughly speaking, since e of $\mathsf{fold}(\lambda a.\lambda r.r)(e)$ represents a graph, the recursive call of $\mathsf{fold}(\lambda a.\lambda r.r)$ must take the same argument twice in the evaluation. Memoization is used to detect the situation, and make the call result in the black hole.

5.1 Modified Syntax with Memos

As mentioned above, we adopt memoization for structural recursions. Accordingly, the syntax of FUnCAL is changed as

$$e ::= \cdots \mid \mathsf{fold}^M e$$

where $\mathsf{fold}_n e$ is replaced with $\mathsf{fold}^M e$ in which M represents a memo. The memos M are all $\emptyset$ initially, and entries are added through evaluation. Although tuples and projections are important in previous sections, we shall ignore them henceforth because they are not relevant in our technical development in the following sections; our discussions can be extended to them straightforwardly. This is the reason why $\mathsf{fold}^M e$ above does not have the subscript.

In what follows, we shall use a metavariable C for binary constructors ":" and "$\cup$".

$$
\begin{array}{ll}
\langle E[x] \mid x = e, \mu\rangle & \to \langle E[x := e] \mid x = \bullet, \mu\rangle \\
\langle E[x := v] \mid x = \bullet, \mu\rangle & \to \langle E[v] \mid x = v, \mu\rangle \\
\langle E[(\lambda x.e_1)\ e_2] \mid \mu\rangle & \to \langle E[e_1] \mid x = e_2, \mu\rangle \\
\langle E[\bullet\ e_2] \mid \mu\rangle & \to \langle E[\bullet] \mid \mu\rangle \\
\langle E[\mathsf{C}\ e_1\ e_2] \mid \mu\rangle & \to \langle E[\mathsf{C}\ x_1\ x_2] \mid x_1 = e_1, x_2 = e_2, \mu\rangle \\
& \qquad (x_1, x_2\text{: fresh}) \\
\langle E[\mathsf{fix}_{\mathbf{G}}\ e] \mid \mu\rangle & \to \langle E[w] \mid w = e\ w, \mu\rangle \quad (w\text{: fresh}) \\
\langle E[\mathsf{fold}^M e\ \bullet] \mid \mu\rangle & \to \langle E[\bullet] \mid \mu\rangle \\
\langle E[\mathsf{fold}^M e\ v] \mid \mu\rangle & \to \langle E[w] \mid w = e\ x_1\ (\mathsf{fold}^{M'} e\ x_2), \mu\rangle \\
& \qquad (M' = M[v \mapsto w], w\text{: fresh}) \\
& \qquad \text{if } v = x_1 : x_2, v \notin \mathsf{dom}(M) \\
\langle E[\mathsf{fold}^M e\ v] \mid \mu\rangle & \to \langle E[w] \mid w = \mathsf{fold}^{M'} e\ x_1 \cup \mathsf{fold}^{M'} e\ x_2, \mu\rangle \\
& \qquad (M' = M[v \mapsto w], w\text{: fresh}) \\
& \qquad \text{if } v = x_1 \cup x_2, v \notin \mathsf{dom}(M) \\
\langle E[\mathsf{fold}^M e\ v] \mid \mu\rangle & \to \langle E[w] \mid \mu\rangle \\
& \qquad \text{if } M(v) = w
\end{array}
$$

Figure 7. Reduction rules of our lazy abstract machine: we assume that $(\lambda x.e)$ is α-renamed so that the variables introduced in the right-hand sides are fresh.

5.2 Semantics

Now, we describe our lazy semantics to execute FUnCAL programs as graph transformations.

A *value* v, or weak head normal form, is defined by:

$$v ::= \mathsf{a} \mid \mathsf{C}\ x_1\ x_2 \mid \lambda x.e \mid \mathsf{fold}^M e \mid \bullet.$$

That is, a value is either of a label, an expression guarded by a constructor C where x_1 and x_2 refer some expressions via a heap introduced later, a function, a memoized recursion, or the black hole. An *evaluation context* E is defined by:

$$E ::= \Box \mid E\ e \mid (\mathsf{fold}^M e)\ E \mid x := E$$

An evaluation context $x := \Box$ is characteristic in lazy evaluation, which represents heap-update after the expression referred by x becomes a value. We write $E[e]$ for an expression obtained from E by replacing $\Box$ with e. Note that $(\mathsf{fold}^M e)\ E$ above means that $\mathsf{fold}^M e$ is strict. A *heap* is a mapping from variables to expressions. A *configuration* is a pair $\langle x \mid \mu\rangle$ where x is a variable to hold an expression to be evaluated in μ and μ is a heap. We assume that configurations are closed; $\langle x \mid \mu\rangle$ is *closed* if x and every variable occurring in the right-hand sides of μ also occur in a left-hand side of μ. One can think that $\langle x \mid x_1 = e_1, \ldots, x_n = e_n\rangle$ as **letrec** $x_1 = e_1$ **and** $\ldots$ **and** $x_n = e_n$ **in** x, but we use different notation to avoid confusion with our restricted forms of recursive definitions such as $\mathsf{fix}_{\mathbf{G}}$ and fold. We sometimes write $\langle e \mid \mu\rangle$ for $\langle x \mid x = e, \mu\rangle$ where we do not care x. We assume that expressions are appropriately α-renamed to avoid capturing; more explicit treatment of variable renaming can be found in [34].

The reduction relation $\langle x \mid \mu\rangle \to \langle x \mid \mu'\rangle$ is defined by the rules in Figure 7. The rules except the ones for $\mathsf{fold}^M e$ are just straightforward extensions of [34] with constructors.

The black hole $\bullet$ represents the apparent undefinedness yielded when the value of x is required by the evaluation of x itself [34]. A typical example is $\mathsf{fix}_{\mathbf{G}}\ (\lambda x.x)$, which will be evaluated as:

$$
\begin{array}{rll}
\langle \mathsf{fix}_{\mathbf{G}}\ (\lambda x.x) \mid \emptyset\rangle & \to & \langle w \mid w = (\lambda x.x)\ w\rangle \\
& \to & \langle w := (\lambda x.x)\ w \mid w = \bullet\rangle \\
& \to & \langle w := x \mid w = \bullet, x = w\rangle \\
& \to & \langle w := (x := w) \mid w = \bullet, x = \bullet\rangle \\
& \to^* & \langle w := (x := \bullet) \mid w = \bullet, x = \bullet\rangle \\
& \to^* & \langle \bullet \mid w = \bullet, x = \bullet\rangle
\end{array}
$$

In contrast, $\mathsf{fix}_{\mathbf{G}}\ (\lambda x.\mathsf{a} : x)$ does not lead to $\bullet$ because of the lazy semantics; recall that the values are weak head normal forms.

$$
\begin{array}{l}
\langle \mathsf{fix}_{\mathbf{G}}\ (\lambda x.\mathsf{a} : x) \mid \emptyset\rangle \\
\to \langle w \mid w = (\lambda x.\mathsf{a} : x)\ w\rangle \\
\to \langle w := ((\lambda x.\mathsf{a} : x)\ w) \mid w = \bullet\rangle \\
\to \langle w := \mathsf{a} : x \mid x = w, w = \bullet\rangle \\
\to \langle w := a' : x' \mid a' = \mathsf{a}, x' = x, x = w, w = \bullet\rangle \\
\to \langle a' : x' \mid a' = \mathsf{a}, x' = x, x = w, w = a' : x'\rangle
\end{array}
$$

The reduction rules for $\mathsf{fold}^M e$ are keys in this semantics. It basically works as $\mathsf{fold}_1 e$ in Section 4.1; indeed, the first, the second and the third rules correspond to (Fold1), (Fold2) and (Fold3), respectively. The first rule also says that $\bullet$ can be seen as an exception. The second and the third rules update the memo, and the fourth rule of $\mathsf{fold}^M e$ looks up the memo if there is corresponding entry in the memo.

Thanks to memoization, $\mathsf{fold}^{\emptyset}(\lambda z.\lambda r.r)\ (\mathsf{fix}_{\mathbf{G}}(\lambda x.\mathsf{a} : x))$, the problematic example shown in Section 1, evaluates to $\bullet$ without major changes to the original semantics [34]. For an illustration, see Example 1 below.

Example 1 (Eliminate All Edges)**.** Let us consider the expression

$$\mathsf{fold}^{\emptyset}(\lambda z.\lambda r.r)\ (\mathsf{fix}_{\mathbf{G}}(\lambda x.\mathsf{a} : x))$$

Here, $\mathsf{fold}^{\emptyset}(\lambda z.\lambda r.r)$ eliminates all the edges, and thus we expect that the expression results in $\bullet$ (a singleton graph). Let us write el_M for $\mathsf{fold}^M(\lambda z.\lambda r.r)$, then the above expression is evaluated as follows.

$$
\begin{array}{ll}
\multicolumn{2}{l}{\langle el_{\emptyset}\ (\mathsf{fix}_{\mathbf{G}}(\lambda x.\mathsf{a} : x)) \mid \emptyset\rangle} \\
\to^* & \underline{\langle el_{\emptyset}\ (a' : x') \mid a' = \mathsf{a}, x' = x, x = w, w = a' : x', \ldots\rangle} \\
\to & \langle u \mid u = (\lambda z.\lambda r.r)\ a'\ (el_{\{(a' : x') \mapsto u\}} x'), \ldots\rangle \\
\to^* & \langle u := el_{\{(a' : x') \mapsto u\}}\ x' \mid u = \bullet, \ldots\rangle \\
\to^* & \underline{\langle u := el_{\{(a' : x') \mapsto u\}}\ (a' : x') \mid u = \bullet, \ldots\rangle} \\
\to & \langle u := u \mid u = \bullet, \ldots\rangle \quad \{\text{ hit! }\} \\
\to & \langle \bullet \mid \ldots\rangle
\end{array}
$$

Here, we underlined the configurations where el_M inspected the memo M. Memoization plays an important role to achieve the intuitive result. At the first-underlined reduction of el_M, the entry $(a' : x') \mapsto u$ is added to M. At the second-underlined reduction of el_M, since $M(a' : x') = u$ holds, the call is reduced to the variable u. Note that, in the evaluation of u after the first-underlined reduction of el_M, the referred value was replaced with $\bullet$. Thus, we got $\bullet$ as the final result. $\Box$

An important property on memos is that, if looking-up succeeds, then the looked-up object must be a value, which will be used in Section 7.

Lemma 1 (Look-up)**.** *Suppose that* $\langle e \mid \emptyset\rangle \to^* \langle E[\mathsf{fold}^M e\ v] \mid \mu\rangle$ *where* $M(v) = x$*. Then,* $\mu(x)$ *is a value.* $\Box$

5.3 Extracting Graphs

As mentioned early in this section, we obtain a graph from an evaluation result as the "graph" structure of a heap. For example, for a configuration $\langle x \mid x = a : x, a = \mathsf{a}\rangle$, we obtain a graph $G = (V, E, I, O)$ with $V = \{x\}$, $E = \{(x, \mathsf{a}, x)\}$, $I = \{\& \mapsto x\}$ and $O = \emptyset$. In general, since a heap may contain unevaluated expressions, we have to evaluate them to extract a graph from the heap. Here, we formally describe how to extract a graph from a heap. In this subsection and in Section 7, we shall omit $(\cup)$ for simplicity.

If a heap contains unevaluated expressions as $\{x = (\lambda y.y)(\mathsf{a} : x)\}$, we cannot extract a graph directly from the heap. To extract a graph from a configuration $\langle x \mid \mu\rangle$, we have to ensure that $\mu(y)$ is a value for all y accessible from the root x. Formally, we say that a variable

x is *accessible* from y in μ when (x, y) in the reflexive transitive closure of the $\{(z, w) \mid w \in \mathrm{fv}(\mu(z))\}$.

For a configuration $\langle x \mid \mu \rangle$, this deep evaluation is easily done by evaluating $\langle \mathsf{elim}_\emptyset\ x \mid \mu \rangle$ where $\mathsf{elim}_M = \mathsf{fold}^M(\lambda a.\lambda r.\mathbf{if}\ a = a\ \mathbf{then}\ r\ \mathbf{else}\ r)$. An application $\mathsf{elim}_\emptyset\ x$ eliminates all the edges from x, and thus it results in $\bullet$ if terminates. If $\langle \mathsf{elim}_\emptyset\ x \mid \mu \rangle \to^* \langle \bullet \mid \mu' \rangle$, then $\langle x \mid \mu \rangle$ and $\langle x \mid \mu' \rangle$ are "bisimilar", and $\langle x \mid \mu' \rangle$ satisfies the required condition above.

Now, let us define how to extract a graph from a heap. Let e_0 be a closed expression of type G, and suppose that we have $\langle \mathsf{elim}_\emptyset\ x_0 \mid x_0 = e_0 \rangle \to^* \langle \bullet \mid \mu \rangle$. Then, from the discussions above, $\mu(x)$ is a value for every x accessible from x_0. We can easily construct a graph from such a heap by $\mathsf{graphify}(x_0, e_0)$ defined as follows.

$$
\begin{array}{l}
\mathsf{graphify}(x_0, e_0) = (V, E, \{\& \mapsto x_0\}, \emptyset) \\
\quad \text{where}\quad V = \text{accessible variables from } x \text{ in } \mu \\
\qquad\qquad\ \ E = \bigcup_{x=(x_1:x_2)\in\mu, x\in V} \{(x, \mu(x_1), x_2)\} \\
\qquad\qquad\ \ \langle \mathsf{elim}_\emptyset\ x_0 \mid x_0 = e_0 \rangle \to^* \langle \bullet \mid \mu \rangle
\end{array}
$$

Note that $\mu(x_1)$ above is a value, more concretely a label literal.

Thus, the termination under the context $\mathsf{elim}_\emptyset\ \square$ means that an expression corresponds to a finite graph.

Treatment of $(\cup)$**.** If we have $(\cup)$, it suffices to use a context $\mathsf{isEmpty}^\emptyset\ (\mathsf{elim}_\emptyset \square)$ instead of $\mathsf{elim}_\emptyset \square$, where $\mathsf{isEmpty}^M$ is a memoized FUnCAL version of $\mathbf{isEmpty}$. If we only allow $\mathsf{isEmpty}^\emptyset$ to appear the outermost context, only an extra effort to prove the termination is one more case analysis added to the proof of Lemma 9. With $(\cup)$, the definition of $\mathsf{graphify}(x_0, e_0)$ is changed accordingly; specifically, the definition of E is changed to $E = \cdots \cup \bigcup_{x=(x_1 \cup x_2)\in\mu, x\in V} \{(x, \varepsilon, x_1), (x, \varepsilon, x_2)\}$.

The next theorem states that the two semantics (the call-by-name and the lazy abstract machine) coincide.

Theorem 2. *If* $\mathsf{graphify}(x, e) = G$*,* G_e *and* G *are bisimilar.*

Proof (Sketch). We extend the notation of the reduction graph to the configurations, and show that reductions of the abstract machine preserves the reduction graphs (up to bisimilarity). For $\langle x \mid \mu \rangle$ where $\mu(y)$ is a value for any y accessible from x, it is easy to prove that the reduction graph and the result of $\mathsf{graphify}$ are bisimilar. $\square$

Remark. This construction of a graph from a configuration runs in time linear to the heap size. This efficient construction is achieved by ε-edges that postpone $\cup$-operation. To obtain ε-free graphs, we have to pay a similar cost to ε-elimination in automata, i.e., cubic time to the number of nodes (= the size of the heap).

Together with Lemma 1, the following lemma says that growth of memo M of elim_M does not change the termination, which will be used in Section 7.

Lemma 2 (Memo and Termination)**.** *If* $\langle \mathsf{elim}_\emptyset\ e \mid \mu \rangle$ *terminates, then* $\langle \mathsf{elim}_M\ e \mid \mu \rangle$ *also terminates for any* M *such that* $\mu(M(v))$ *is a value for all* $v \in \mathsf{dom}(M)$*.* $\square$

6. Type System

In this section, we describe the type system that guarantees termination of $\to$ under the context $\mathsf{elim}_\emptyset \square$; that is, well-typed expressions are finite-graph transformations. We shall only show the type system in this section; termination will be discussed in Section 7.

6.1 Idea

In advance to the formal definition of our type system, we discuss a problematic example we want to exclude to explain the underlying idea of the type system. Consider the expression $aInB\ bs$ where

$$
\begin{array}{ll}
bs & = \mathsf{fix}_{\mathbf{G}}\ (\lambda x.\mathtt{b} : x) \\
aInB & = \mathsf{fold}(\lambda z.\lambda r.z : insA\ r) \\
insA & = \mathsf{fold}(\lambda z.\lambda r.\mathtt{a} : z : r).
\end{array}
$$

(Here, we ignore memos for a while.) One might notice that $insA$ is applied to a variable r that holds the result of the recursive call of $aInB$. The expression evaluates to a nonregular tree as

$$
\begin{array}{ll}
aInB\ bs & \to^* \mathtt{b} : insA\ (aInB\ bs) \\
& \to^* \mathtt{b} : insA\ (\mathtt{b} : insA\ (aInB\ bs)) \\
& \to^* \mathtt{b} : \mathtt{a} : \mathtt{b} : insA^2\ (aInB\ bs) \\
& \to^* \mathtt{b} : \mathtt{a} : \mathtt{b} : \mathtt{a} : \mathtt{a} : \mathtt{a} : \mathtt{b} : insA^3\ (aInB\ bs) \\
& \to^* \mathtt{b} : \mathtt{a} : \mathtt{b} : \mathtt{a} : \mathtt{a} : \mathtt{a} : \mathtt{b} : \cdots : insA^n\ (aInB\ bs)
\end{array}
$$

and thus must not correspond to a finite graph. Here, one can find that the number of nested applications of $insA$ increases in the evaluation, which leads to this nonregularity and thus nontermination of $\mathsf{elim}_\emptyset\ (aInB\ bs)$. In contrast, such nonregularity does not arise for functions $insA$ itself and el in Example 1. For example, if we apply them to bs above, we have

$$
el\ bs \to^* el\ bs \qquad\qquad insA\ bs \to^* \mathtt{a} : \mathtt{b} : insA\ bs
$$

Thanks to this looping structure, $\mathsf{elim}_\emptyset\ (el\ bs)$ and $\mathsf{elim}_\emptyset\ (insA\ bs)$ terminate with memoization.

To exclude such a problematic case, we use a modal type system with modality $\bigcirc$ that represents "already constructed (and thus regular)", and restrict that the argument of $\mathsf{fold} f$ to be a tree that is already constructed and regular. For an expression e of type $\bigcirc\mathsf{G}$, since we know that e is evaluate to a regular tree, the application of $\mathsf{fold} f$ to e terminates, as $insA\ bs$ where we know the regularity of bs *beforehand* the application. However, for an expression e of type G, application of $\mathsf{fold} f$ is not always terminating because its results can be referred in the e by some outer-contexts, as $insA\ bs$ and its simpler version $\mathsf{fix}_{\mathbf{G}}(\lambda x.\mathtt{b} : insA\ x)$. Here we cannot know the regularity of the argument because the tree is being constructed.

6.2 Modal Types

A *type* τ is defined as follows.

$$
\tau ::= \mathcal{G} \mid \tau_1 \to \tau_2 \mid \mathsf{L} \qquad \mathcal{G} ::= \mathsf{G} \mid \bigcirc\mathcal{G}
$$

Types consist of graph types with modality ($\mathcal{G}$), function types ($\to$) and the label type (L). As explained above, the modality $\bigcirc$ represents "already constructed". For example, the argument of fold must has type $\bigcirc\mathcal{G}$ to produce a graph of type $\mathcal{G}$. It is natural to have a subtyping relation $\bigcirc\mathcal{G} \preceq \mathcal{G}$, which means that a graph constructed in some period is still available after the period. The structural subtyping rules for $\preceq$ are standard ones and we shall omit them.

Figure 8 shows the typing rules. The typing rules for variables and λs are standard ones. The rules T-CONS, T-CHOICE and T-FIX says that graph constructors construct a graph in a period from graphs in the same period. The rule T-BH says that $\bullet$ is something similar to an exception. The rule T-CATA is special in our type system. The rule requires that the argument of fold must be a graph that is constructed beforehand; for such a tree, we can use its finiteness and guarantee the termination of the application. In the premise of T-CATA, since the memo maps arguments of $\mathsf{fold}^M e$ to their return values, v must be of type $\bigcirc\mathcal{G}$ and $M(v)$ must be of type $\mathcal{G}$.

Our type system can be easily extended to configurations, and we can easily prove that the preservation and the progress property.

An important property for our purpose is that functional programs translated from UnCAL as in Section 4 are well-typed.

Theorem 3. *If* $\vdash g :: \mathsf{DB}_Y^X \rightsquigarrow e$*, then* $\vdash e :: (\bigcirc^n \mathsf{G})^{|Y|} \to (\bigcirc^n \mathsf{G})^{|X|}$ *for some* n*.*

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x :: \tau}\text{T-VAR} \quad \frac{\Gamma, x :: \tau_1 \vdash e :: \tau_2}{\Gamma \vdash \lambda x.e :: \tau_1 \to \tau_2}\text{T-ABS}$$

$$\frac{\Gamma \vdash e_1 :: \tau' \to \tau \quad \Gamma \vdash e_2 :: \tau'}{\Gamma \vdash e_1\, e_2 :: \tau}\text{T-APP}$$

$$\frac{}{\Gamma \vdash \mathsf{a} :: \mathsf{L}}\text{T-LABEL} \quad \frac{\Gamma \vdash e_1 :: \mathsf{L} \quad \Gamma \vdash e_2 :: \mathcal{G}}{\Gamma \vdash e_1 : e_2 :: \mathcal{G}}\text{T-CONS}$$

$$\frac{\{\Gamma \vdash e_i :: \mathcal{G}\}_{i=1,2}}{\Gamma \vdash e_1 \cup e_2 :: \mathcal{G}}\text{T-CHOICE} \quad \frac{}{\Gamma \vdash \bullet :: \tau}\text{T-BH}$$

$$\frac{\Gamma \vdash e :: \mathcal{G} \to \mathcal{G}}{\Gamma \vdash \mathsf{fix}_{\mathbf{G}}\, e :: \mathcal{G}}\text{T-FIX}$$

$$\frac{\begin{array}{c}\Gamma \vdash e :: \mathsf{L} \to \mathcal{G} \to \mathcal{G} \\ \{\Gamma \vdash v :: \bigcirc\mathcal{G} \wedge \Gamma \vdash x :: \mathcal{G} \mid v \in \mathsf{dom}(M), x = M(v)\}\end{array}}{\Gamma \vdash \mathsf{fold}^M e :: \bigcirc\mathcal{G} \to \mathcal{G}}\text{T-CATA}$$

$$\frac{\Gamma \vdash e :: \tau' \quad \tau' \preceq \tau}{\Gamma \vdash e :: \tau}\text{T-SUB}$$

Figure 8. Typing rules for termination.

Proof. See Appendix B. □

We state that the typed expressions respect bisimulation; in other words, a typed expression cannot distinguish bisimilar expressions (in the sense of Section 4.2).

Theorem 4. *Suppose* $\vdash f :: \mathcal{G} \to \mathcal{G}'$. *For any* e_1 *and* e_2 *such that* $\vdash e_i :: \mathcal{G}$ $(i = 1, 2)$, *if* $e_1 \sim e_2$, *then* $f\, e_1 \sim f\, e_2$.

Proof. See Appendix C. □

One might think that it would suffice to have only two graph types $\bigcirc\mathsf{G}$ and G instead of having many graph types $\bigcirc^n\mathsf{G}$ would suffice because $\bigcirc$ represents capability to be an argument of fold, and have a type rule $\mathsf{fold} :: (\mathsf{L} \to \mathsf{G} \to \mathsf{G}) \to \bigcirc\mathsf{G} \to \mathsf{G}$. However, this prohibits the composition of fold, which means that we cannot express what the original UnCAL can express. One then might think that we could use the rule $\mathsf{fold} :: (\mathsf{L} \to \mathsf{G} \to \mathsf{G}) \to \bigcirc\mathsf{G} \to \bigcirc\mathsf{G}$ instead. However, this violates the type safety; note that we have $\lambda x.\mathsf{fold}(\lambda a.\lambda r.x)\, (\mathtt{Dummy} : \bullet) :: \mathsf{G} \to \bigcirc\mathsf{G}$.

7. Soundness of the Type System

In this section, we prove that every expression e of type G represents a finite graph; i.e., the evaluation of $\mathsf{elim}_\emptyset\, e$ terminates and thus graphify succeeds. To simplify our discussion, as wrote in Section 5.3, we ignore $(\cup)$ in this section.

Formally, we prove the following property:

Theorem 5 (Soundness)**.** *If* $\vdash e :: \mathsf{G}$, *then* $\langle \mathsf{elim}_\emptyset\, x \mid x = e\rangle \to^* \langle \bullet \mid \mu\rangle$.

We will prove the theorem by using logical relation [39].

7.1 Modification to Type System

Recall that we have said that our type system can be extended to heaps and configurations; a solution would be defining that μ is well-typed under Γ if $\forall x.\ \Gamma \vdash \mu(x) :: \Gamma(x)$. However, typing derivations could be cyclic via μ in this naive approach. This prevents us from using the logical-relation based termination proof as in the simply-typed λ-calculus [39]. In other words, a configuration $\langle x \mid \mu\rangle$ is essentially cyclic [2].

To overcome the problem, we parameterize a type system by a heap. In analogy with the lazy evaluation strategy in which every variable is evaluated at most once, every variable is dereferenced at most once in typing. This idea is realized by splitting T-VAR into the following two rules.

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash_\mu x :: \tau}\text{T-VAR} \quad \frac{\Gamma \vdash_{\mu, x=\bullet} e :: \tau}{\Gamma \vdash_{\mu, x=e} x :: \tau}\text{T-VARREC}$$

Here, we assume that $\mathsf{dom}(\mu)$ and $\mathsf{dom}(\Gamma)$ are disjoint, which means that there is no nondeterminism to apply T-VAR or T-VARREC. The other rules remain unchanged.

The following fact says that an expression and a heap that are typable in the original type system are also typable in the parameterized type system.

Lemma 3. *If we have* $\Gamma, \Delta \vdash e :: \tau$ *and* $\Gamma, \Delta \vdash \mu(x) :: \Delta(x)$ *for any* x *in* $\mathsf{dom}(\Delta)$, *then* $\Gamma \vdash_\mu e :: \tau$ *holds.* □

We define a *substitution* σ as a mapping from variables to expressions of which domain is finite. We write $t\sigma$ for the application of the substitution σ to an expression/heap t.

The following lemma says that an evaluation of an expression of type $\mathcal{G}$ cannot "observe" graphs of type $\mathcal{G}$ because of the modality restriction; recall that fold is the only language construct that can observe graphs.

Lemma 4. *Let* $Z = \{z_1, \ldots, z_k\}$. *Suppose* $z_1 :: \mathcal{G}, \ldots, z_k :: \mathcal{G} \vdash_\mu e :: \mathcal{G}$. *Then, if* $\langle \mathsf{elim}_M\, e\sigma \mid \mu\sigma\rangle$ *terminates for some* $\sigma : Z \to V_M$, *then* $\langle \mathsf{elim}_M\, e\sigma' \mid \mu\sigma'\rangle$ *terminates for all* $\sigma' :: Z \to V_M$ *where* $V_M = \mathsf{dom}(M) \cup \{\bullet\}$.

Proof. See Appendix D. □

7.2 Logical Relation

Then, we define a logical relation $\mathcal{R}$.

Definition 2 (Relation $\mathcal{R}$)**.** The unary relation $\mathcal{R}_\tau$ on configurations is defined as follows.

- $\langle e \mid \mu\rangle \in \mathcal{R}_\mathsf{L}$ iff $\langle e \mid \mu\rangle \to^* \langle v \mid \mu'\rangle$ for some v and μ'.
- $\langle e \mid \mu\rangle \in \mathcal{R}_\mathsf{G}$ iff $\langle \mathsf{elim}_\emptyset\, e \mid \mu\rangle \to^* \langle \bullet \mid \mu'\rangle$ for some μ'.
- $\langle e \mid \mu\rangle \in \mathcal{R}_{\bigcirc\mathcal{G}}$ iff $\langle e \mid \mu\rangle \in \mathcal{R}_\mathcal{G}$.
- $\langle f \mid \mu\rangle \in \mathcal{R}_{\tau_1 \to \tau_2}$ iff $\langle f \mid \mu\rangle \to^* \langle v \mid \mu'\rangle$ for some v and μ', and $\langle f\, e \mid \mu \cup \eta\rangle \in \mathcal{R}_{\tau_2}$ for any $\langle e \mid \eta\rangle \in \mathcal{R}_{\tau_1}$.

For heaps μ and μ', we write $\mu \cup \mu'$ for the union of μ and μ', assuming that $\mu(x) = \mu'(x)$ for any $x \in \mathsf{dom}(\mu) \cap \mathsf{dom}(\mu')$. Intuitively, $R(\tau)$ defines pairs of expressions and heaps that are "meaningful" as *finite-graph* transformations. Especially, $\langle e \mid \mu\rangle \in \mathcal{R}_\mathcal{G}$ means that $\langle e \mid \mu\rangle$ corresponds to a finite graph. Note that, thanks to $\mathsf{elim}_\emptyset$ in the definition, we have $aInB\ bs \notin \mathcal{R}_\mathcal{G}$ for $bs = \mathsf{fix}_\mathbf{G}\, (\lambda x.\mathsf{b} : x)$ while the evaluation of $aInB\ bs$ itself terminates. Also note that we have $\mathcal{R}_\mathcal{G} = \mathcal{R}_{\bigcirc\mathcal{G}}$ because $\vdash_\mu e :: \mathcal{G}$ if and only if $\vdash_\mu e :: \bigcirc\mathcal{G}$; the modality is used only to prove Lemma 4.

In the later proof, we will use the following properties on $\mathcal{R}$. Lemma 5 says that "garbage cells" in the heap does not affect termination. Lemma 6 says that one-step reduction does not change the termination property, which is rather obvious.

Lemma 5. $\langle e \mid \mu\rangle \in \mathcal{R}_\tau$ *implies* $\langle e \mid \mu \cup \mu'\rangle \in \mathcal{R}_\tau$. □

Lemma 6 (Preservation under Reduction)**.** *Suppose* $\vdash_\mu E[e] :: \tau$ *and* $\langle E[e] \mid \mu\rangle \to \langle E'[e'] \mid \mu'\rangle$. *Then,* $\langle E[e] \mid \mu\rangle \in \mathcal{R}_\tau$ *if and only if* $\langle E'[e'] \mid \mu'\rangle \in \mathcal{R}_\tau$. □

7.3 Lemmas for Recursive Definitions

In advance to the proof of the termination, we prove some lemmas stating $\mathcal{R}$ is preserved in our recursive definitions.

The following lemma intuitively says that typed $\mathsf{fix}_\mathbf{G}\, e$ expressions corresponds to a finite graph if e preserves finiteness.

Lemma 7 ($\mathsf{fix}_\mathbf{G}$)**.** *Suppose* $\vdash_\mu e_0 :: \mathcal{G} \to \mathcal{G}$ *and* $\langle e_0\, e' \mid \mu \cup \mu'\rangle \in \mathcal{R}_\mathcal{G}$ *for any* $\langle e' \mid \mu'\rangle \in \mathcal{R}_\mathcal{G}$. *Then, we have* $\langle \mathsf{fix}_\mathbf{G}\, e_0 \mid \mu\rangle \in \mathcal{R}_\mathcal{G}$.

Proof. It suffices to show that $\langle \mathsf{elim}_\emptyset\ w \mid w = e_0\ w, \mu\rangle$ terminates. Consider a configuration $c_0 = \langle \mathsf{elim}_\emptyset\ w \mid w = e_0\ u, u = \bullet, \mu\rangle$ which means $\mathsf{fix}_{\mathbf{G}}$ is unfolded only once. We prove the statement by showing that $d_0 = \langle \mathsf{elim}_\emptyset\ w \mid w = e_0\ w, \mu\rangle$ terminates if c_0 terminates. Then, since we have the termination of c_0 by applying the premise of this lemma twice, we conclude the termination of d_0.

It is easy to prove that, for any e that is not u, $\langle E[e] \mid \eta_*\rangle \rightarrow \langle E'[e'] \mid \eta'_*\rangle$ if and only if $\langle E[e][w/u] \mid \eta\rangle \rightarrow \langle E'[e'][w/u] \mid \eta\rangle$. Here, η_* is the heap $(w = e[w/u], u = \bullet, \eta')$ for a heap $\eta = (w = e, \eta')$. Thus, to compare the evaluation sequences from c_0 and d_0, we focus on how u are evaluated in the sequence from c_0.

Let us consider the reduction sequence from c_0. There are two possibilities about the sequence in whether u is evaluated. If u is not evaluated, the reductions from c_0 and d_0 are clearly bisimilar, and thus d_0 terminates. If u is evaluated, the reduction sequences may differ from when u is evaluated for the first time.

Let us consider a reduction sequence from c_0 in which u is evaluated. It must have the form of:

$$c_0 \rightarrow^* \langle \mathsf{elim}_M\ (E[u]) \mid \eta_*\rangle \rightarrow^* \dots$$

Accordingly, we must have the following sequence.

$$d_0 \rightarrow^* \langle \mathsf{elim}_M\ (E[w/u][w]) \mid \eta\rangle \dots$$

On the one hand, since $\eta_*(u) = \bullet$ by the definition, we have

$$\langle \mathsf{elim}_M\ (E[u]) \mid \eta_*\rangle \rightarrow^* \langle \mathsf{elim}_M\ (E[\bullet]) \mid \eta_*\rangle.$$

On the other hand, we have $\eta(w) = v$ for some v because w has been evaluated beforehand. Thus, we must have

$$\langle \mathsf{elim}_M\ (E[w/u][w]) \mid \eta\rangle \rightarrow^* \langle \mathsf{elim}_M\ (E[w/u][v]) \mid \eta\rangle.$$

Since $\bullet, v \in \mathsf{dom}(M)$, from Lemma 4, the reductions from $\langle \mathsf{elim}_M\ (E[\bullet]) \mid \eta_*\rangle$ and $\langle \mathsf{elim}_M\ (E[w/u][v]) \mid \eta\rangle$ terminate if either one terminates. That is, c_0 terminates if and only if d_0 terminates. Since c_0 terminates, d_0 also terminates. $\square$

The following lemma states that every typed $\mathsf{fold}^M e$ results in a finite graph if it is applied to an expression that results in a finite graph. Note that, we use the finiteness of the argument in this proof.

Lemma 8 (fold)**.** *Suppose we have $\vdash_\mu \mathsf{fold}^M e :: \bigcirc\mathcal{G} \rightarrow \mathcal{G}$ and we have $\langle e \mid \mu\rangle \in \mathcal{R}_{\mathsf{L}\rightarrow\mathcal{G}\rightarrow\mathcal{G}}$. Then, $\langle \mathsf{fold}^M e \mid \mu\rangle \in \mathcal{R}_{\bigcirc\mathcal{G}\rightarrow\mathcal{G}}$.*

Proof (Sketch). We prove that $\langle \mathsf{fold}^M e\, e' \mid \mu \cup \mu'\rangle \in \mathcal{R}_{\mathcal{G}}$ holds for any $\langle e' \mid \mu'\rangle \in \mathcal{R}_{\bigcirc\mathcal{G}}$.

The statement will be proved in three steps:

1. We prove the termination of $\mathsf{fold}_{(k)}e$, where the number of applications is limited by k and the memoization is not exploited.
2. We prove the termination of $\mathsf{fold}^M_{(k)}e$, where the number of applications is limited by k, but memoization is exploited.
3. We prove the termination of $\mathsf{fold}^M e$.

For Step 1, we introduce a new language construct $\mathsf{fold}_{(k)}e$ to limit the number of recursions. Concretely, for $k > 0$, its evaluation rules are similar to those of fold^M, except that $\mathsf{fold}^{M'}$s in the RHSs are replaced with $\mathsf{fold}_{(k-1)}$ and $\mathsf{fold}_{(k)}$ does not use memoization. For $k = 0$, its evaluation rule is as follows.

$$\langle E[\mathsf{fold}_{(0)} e\ v] \mid \mu\rangle \rightarrow \langle E[\bullet] \mid \mu\rangle$$

By the induction on k, we can prove that $\langle \mathsf{fold}_{(k)} e\, e' \mid \mu \cup \mu'\rangle \in \mathcal{R}_{\mathcal{G}}$ for any k. The types, or more precisely the modality $\bigcirc$, are not relevant in this proof; even $\mathsf{elim}_\emptyset\ (aInB\ bs)$ terminates if we replace $\mathsf{fold}^\emptyset$ with $\mathsf{fold}_{(k)}$ in the definition of $aInB$.

For Step 2, similar to Step 1, we introduce a new language construct $\mathsf{fold}^M_{(k)}e$ which has the similar semantics to $\mathsf{fold}_{(k)}e$ but it looks up memo as $\mathsf{fold}^M e$ does. A key observation is that for any configuration $\langle E[\mathsf{fold}^M_{(k)} e\ v] \mid \mu\rangle$, if $M(v) = x$, then $\mu(x)$ is a value from Lemma 1. Thus, from Lemma 4, we can prove that $\langle \mathsf{elim}_\emptyset\ (\mathsf{fold}^M_{(k)} e\, e') \mid \mu \cup \mu'\rangle$ terminates if and only if $\langle \mathsf{elim}_\emptyset\ (\mathsf{fold}_{(k)} e\, e') \mid \mu \cup \mu'\rangle$ terminates. Note that, since the modality information is used here to apply Lemma 4, the same discussion cannot be applied to $aInB$.

For Step 3, we show that there exists some k_0 such that $\langle \mathsf{elim}_\emptyset\ (\mathsf{fold}^M_{(k)} e\, e') \mid \mu \cup \mu'\rangle$ terminates for some $k \geq k_0$ if and only if $\langle \mathsf{elim}_\emptyset\ (\mathsf{fold}^M e\, e') \mid \mu \cup \mu'\rangle$ terminates. Since we have $\langle e' \mid \mu'\rangle \in \mathcal{R}_{\bigcirc\mathcal{G}}$, we have that $\langle \mathsf{elim}_\emptyset e' \mid \mu'\rangle$ terminates. Then, we can show that v' that occurs as $\langle \mathsf{elim}_\emptyset\ (\mathsf{fold}^M_{(k)} e\, e') \mid \mu \cup \mu'\rangle \rightarrow^* \langle E[\mathsf{fold}^{M'}_{(k')} e\ v'] \mid \_\rangle$ also occurs as $\langle \mathsf{elim}_M e' \mid \mu'\rangle \rightarrow^* \langle \mathsf{elim}_{M'}\ v' \mid \_\rangle$. From the termination of $\langle \mathsf{elim}_\emptyset\ e' \mid \mu'\rangle$, we can say that the number of arguments of $\mathsf{fold}^M_{(\_)} e$ is at most finite. Thus, thanks to the memoization, let k_0 be the number of such v's, we have that $\langle \mathsf{elim}_\emptyset\ (\mathsf{fold}^M_{(k)} e\, e') \mid \mu \cup \mu'\rangle$ terminates if and only if $\langle \mathsf{elim}_\emptyset\ (\mathsf{fold}^M e\, e') \mid \mu \cup \mu'\rangle$ terminates, for all $k > k_0$. Thus, we have $\langle \mathsf{fold}^M e\, e' \mid \mu \cup \mu'\rangle \in \mathcal{R}_{\mathcal{G}}$. $\square$

7.4 Proof of Termination

Now we are ready to prove the main theorem in this section. To prove the main theorem, we prove the following more general property.

Lemma 9. *Suppose $\langle e_x \mid \mu'\rangle \in \mathcal{R}_{\Gamma(x)}$ for any $x \in \mathsf{dom}(\Gamma)$. If $\Gamma \vdash_\mu e :: \tau$, then $\langle e \mid \mu \cup \mu' \cup \{x = e_x\}_{x \in \mathsf{dom}(\Gamma)}\rangle \in \mathcal{R}_\tau$.*

Proof. We prove the statement by using the induction on the typing derivation. Let η be $\mu \cup \mu' \cup \{x = e_x\}_{x \in \mathsf{dom}(\Gamma)}$. We only show the proofs for non-trivial cases.

Case T-VARREC. In this case, we have $e = x \in \mathsf{dom}(\mu)$. From the induction hypothesis, we have $\langle \mu(x) \mid \mu, x = \bullet\rangle \in \mathcal{R}_\tau$. From Lemma 6, we have $\langle x \mid \mu\rangle \in \mathcal{R}_\tau$. Then, from Lemma 5 we have $\langle x \mid \eta\rangle \in \mathcal{R}_\tau$.

Cases T-CONS. In this case, we have $e = e_1 : e_2$ and $\tau = \mathcal{G}$.

The reduction sequence starting from $\langle \mathsf{elim}_\emptyset\ (e_1 : e_2) \mid \eta\rangle$ must has the form of

$$\begin{array}{l} \langle \mathsf{elim}_\emptyset\ (e_1 : e_2) \mid \eta\rangle \\ \rightarrow^* \langle \mathsf{elim}_\emptyset\ (x_1 : x_2) \mid x_1 = e_1, x_2 = e_2, \eta\rangle \\ \rightarrow^* \langle E[a] \mid a = x_1, r = \mathsf{elim}_M x_2, x_1 = e_1, x_2 = e_2, \eta\rangle \end{array}$$

where $E[a] = \mathbf{if}\ a = a\ \mathbf{then}\ r\ \mathbf{else}\ r$. Since we have $\langle e_1 \mid \eta\rangle \in \mathcal{R}_{\mathsf{L}}$ from the induction hypothesis, we have that the evaluation of a above terminates from Lemmas 5 and 6. Thus, the reduction continues as

$$\begin{array}{l} \langle E[a] \mid a = x_1, r = \mathsf{elim}_{M'} x_2, x_1 = e_1, x_2 = e_2, \eta\rangle \\ \rightarrow^* \langle \mathsf{elim}_{M''} x_2 \mid x_1 = \mathsf{a}, x_2 = e_2, \eta'\rangle \end{array}$$

Since we have $\langle e_2 \mid \eta\rangle \in \mathcal{R}_{\mathcal{G}}$ from the induction hypothesis, we have that $\langle \mathsf{elim}_{M''}\ x_2 \mid x_1 = e_1, x_2 = e_2, \eta'\rangle$ terminates from Lemmas 1, 2, 5 and 6. Thus, the reduction sequence terminates and $\langle e_1 : e_2 \mid \eta\rangle \in \mathcal{R}_{\mathcal{G}}$.

Case T-BH. By induction of τ. Note that, if we apply $\bullet$ of type $\tau_1 \rightarrow \tau_2$ to any value, then we obtain a value $\bullet$ of type τ_2.

Case T-FIX. By Lemma 7

Case T-CATA. By Lemma 8. $\square$

As a consequence, we have obtained Theorem 5, which says that every expression of type $\mathcal{G}$ corresponds to a finite graph, in the sense that $\rightarrow$ terminates under the observation $\mathsf{elim}_\emptyset$ $\square$.

8. Related Work

We discuss graph transformations on graphs up to *bisimilarity*, as UnCAL [8]. So far, many frameworks have been studied for manipulation of graphs up to *equality/isomorphism* from the functional programming community [10, 12–14, 19, 25, 27] and from the database community [1, 9]. Since these frameworks and UnCAL handle the different kinds of data structures, their results are incomparable with those of UnCAL and ours. Using the fact that graphs up to bisimilarity are actually infinite trees, we have shown that we can enjoy functional-style program-manipulation techniques for UnCAL graph transformations (Section 2). Since graph-theoretic properties of a graph are usually do not respect the bisimilarity, any graph-transformation language that respects bisimilarity, such as UnCAL, cannot compute them.

In the listed above, the frameworks [10, 14, 19] focus on cyclic trees instead of general graphs, by using μ-terms (e.g., $\mu x.1 : x$ represents an infinite list of 1) in some abstract syntax representations. One might think that we also can use abstract syntax (i.e., frozen μ-terms) instead of expressions with explicit heaps in our technical development. However, the discussion does not scale to tuples: We want to identify $\pi_1 \mu x.(\pi_2 x, \pi_1 x)$ with the black hole for example, but it is not easy to define reduction rules that can reduce $\pi_1 \mu x.(\pi_2 x, \pi_1 x)$ to the black hole. In contrast, a straightforward extension to tuples works well with [34] and ours: $\pi_1(\mathsf{fix}_{\mathbf{G}}\ (\lambda x.(\pi_2 x, \pi_1 x))$ successfully evaluates to $\bullet$. In addition, even with μ-terms, we have to exclude problematic examples such as $\mu x.\mathtt{b} : insA\ x$.

Hidaka et al. [22] extend UnCAL to ordered graphs. Although we discussed unordered graphs, we believe that our discussion would be applicable to the ordered graphs with some modifications.

Our semantics of FUnCAL is based on Nakata and Hasegawa [34]'s lazy semantics. In the context of term graph rewriting, in which they discuss rewriting of (possibly) infinite terms in $\langle x \mid \mu \rangle$ format, more general reduction strategies in addition to the lazy one have been studied [2, 3]. It is important to discuss whether or not our discussion can be lifted to these reduction strategies such as parallel call-by-value ones [8, 35].

9. Conclusion

We have formalized the translation from UnCAL programs to functional ones so that we can reason about UnCAL programs as functional ones. We also have designed the semantics and the type system of the target language FUnCAL to run the translated functional programs as finite-graph transformations with termination guarantee. We have shown that our result enables us to apply several program-manipulation techniques such as verification and optimization to the graph transformation problem.

We believe that our discussions in this paper would be useful not only for extending UnQL/UnCAL but also for designing a graph transformation language that respects bisimilarity. In this direction, it is interesting to extend the type system toward a language with general recursions. This also enables us to apply more optimizations or other program transformations to the translated programs.

Acknowledgments

We thank Meng Wang, Janis Voigtländer, and Patrik Jansson for their helpful comments on an earlier version of this paper. This work was partially supported by JSPS KAKENHI Grant Number 24700020 and 22800003, the Grand-Challenging Project on the "Linguistic Foundation for Bidirectional Model Transformation" of the National Institute of Informatics.

References

[1] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. L. Wiener. The Lorel query language for semistructured data. *Int. J. on Digital Libraries*, 1(1):68–88, 1997.

[2] Z. M. Ariola and S. Blom. Cyclic lambda calculi. In *TACS*, volume 1281 of *LNCS*, pages 77–106. Springer, 1997.

[3] Z. M. Ariola and J. W. Klop. Equational term graph rewriting. *Fundam. Inform.*, 26(3/4):207–240, 1996.

[4] F. Bancilhon and N. Spyratos. Update semantics of relational views. *ACM Trans. Database Syst.*, 6(4):557–575, 1981.

[5] J. Bézivin, B. Rumpe, A. Schürr, and L. Tratt. Model transformations in practice workshop. In *MoDELS Satellite Events*, volume 3844 of *LNCS*, pages 120–127. Springer, 2005.

[6] P. Buneman, S. Davidson, G. Hillebrand, and D. Suciu. A query language and optimization techniques for unstructured data. In *SIGMOD*, pages 505–516. ACM, 1996.

[7] P. Buneman, S. B. Davidson, M. F. Fernandez, and D. Suciu. Adding structure to unstructured data. In *ICDT*, volume 1186 of *LNCS*, pages 336–350. Springer, 1997.

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

[9] M. P. Consens and A. O. Mendelzon. GraphLog: a visual formalism for real life recursion. In *PODS*, pages 404–416. ACM Press, 1990.

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

[11] U. Dayal and P. A. Bernstein. On the correct translation of update operations on relational views. *ACM Trans. Database Syst.*, 7(3):381–416, 1982.

[12] M. Erwig. Graph algorithms = iteration + data structures? the structure of graph algorithms and a corresponding style of programming. In *WG*, volume 657 of *LNCS*, pages 277–292. Springer, 1992.

[13] M. Erwig. Inductive graphs and functional graph algorithms. *J. Funct. Program.*, 11(5):467–492, 2001.

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

[15] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Trans. Program. Lang. Syst.*, 29(3), 2007.

[16] N. Foster, K. Matsuda, and J. Voigtländer. Three complementary approaches to bidirectional programming. In J. Gibbons, editor, *SSGIP*, volume 7470 of *Lecture Notes in Computer Science*, pages 1–46. Springer, 2010. ISBN 978-3-642-32201-3.

[17] A. J. Gill, J. Launchbury, and S. L. P. Jones. A short cut to deforestation. In *FPCA*, pages 223–232, 1993.

[18] L. Grunske, L. Geiger, and M. Lawley. A graphical specification of model transformations with triple graph grammars. In *ECMDA-FA*, volume 3748 of *LNCS*, pages 284–298. Springer, 2005.

[19] M. Hamana. Initial algebra semantics for cyclic sharing tree structures. *Logical Methods in Computer Science*, 6(3), 2010.

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

[21] S. Hidaka, Z. Hu, K. Inaba, H. Kato, K. Matsuda, K. Nakano, and I. Sasano. Marker-directed optimization of UnCAL graph transformations. In *LOPSTR*, volume 7225 of *LNCS*, pages 123–138. Springer, 2011.

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

[23] S. Hidaka, Z. Hu, K. Inaba, H. Kato, and K. Nakano. GRoundTram: An integrated framework for developing well-behaved bidirectional model transformations. *Progress in Informatics*, (10):131–148, 2013.

[24] K. Inaba, S. Hidaka, Z. Hu, H. Kato, and K. Nakano. Graph-transformation verification using monadic second-order logic. In *PPDP*, pages 17–28. ACM, 2011.

[25] T. Johnsson. Efficient graph algorithms using lazy monolithic arrays. *J. Funct. Program.*, 8(4):323–333, 1998.

[26] F. Jouault and J. Bézivin. KM3: A DSL for metamodel specification. In *FMOODS*, volume 4037 of *LNCS*, pages 171–185. Springer, 2006.

[27] D. J. King and J. Launchbury. Structuring depth-first search algorithms in haskell. In *POPL*, pages 344–354, 1995.

[28] J. Launchbury. A natural semantics for lazy evaluation. In *POPL*, pages 144–154, 1993.

[29] K. Matsuda and M. Wang. Bidirectionalization for free with runtime recording: or, a light-weight approach to the view-update problem. In R. Peña and T. Schrijvers, editors, *PPDP*, pages 297–308. ACM, 2013. ISBN 978-1-4503-2154-9.

[30] K. Matsuda and M. Wang. "Bidirectionalization for free" for monomorphic transformations. *Science of Computer Programming*, 2014. doi: http://dx.doi.org/10.1016/j.scico.2014.07.008. URL `http://www.sciencedirect.com/science/article/pii/S0167642314003323`. DOI: `10.1016/j.scico.2014.07.008`.

[31] L. G. L. T. Meertens. Paramorphisms. *Formal Asp. Comput.*, 4(5):413–424, 1992.

[32] E. Meijer, M. M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *FPCA*, volume 523 of *LNCS*, pages 124–144. Springer, 1991.

[33] R. Milner. *Communicating and mobile systems - the Pi-calculus*. Cambridge University Press, 1999. ISBN 978-0-521-65869-0.

[34] K. Nakata and M. Hasegawa. Small-step and big-step semantics for call-by-need. *J. Funct. Program.*, 19(6):699–722, 2009.

[35] S. Nishimura and A. Ohori. Parallel functional programming on recursively defined data via data-parallel recursion. *J. Funct. Program.*, 9(4):427–462, 1999.

[36] A. Ohori and I. Sasano. Lightweight fusion by fixed point promotion. In *POPL*, pages 143–154. ACM, 2007.

[37] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources. In *ICDE*, pages 251–260. IEEE Computer Society, 1995.

[38] M. H. Sørensen, R. Glück, and N. D. Jones. A positive supercompiler. *J. Funct. Program.*, 6(6):811–838, 1996.

[39] W. W. Tait. Intensional interpretations of functionals of finite type I. *Journal of Symbolic Logic*, 32(2):198–212, June 1967.

[40] H. Unno, N. Tabuchi, and N. Kobayashi. Verification of tree-processing programs via higher-order model checking. In *APLAS*, volume 6461 of *LNCS*, pages 312–327. Springer, 2010.

[41] J. Voigtländer. Bidirectionalization for free! (pearl). In Z. Shao and B. C. Pierce, editors, *POPL*, pages 165–176. ACM, 2009. ISBN 978-1-60558-379-2.

[42] Y. Xiong, D. Liu, Z. Hu, H. Zhao, M. Takeichi, and H. Mei. Towards automatic model synchronization from model transformations. In R. E. K. Stirewalt, A. Egyed, and B. Fischer, editors, *ASE*, pages 164–173. ACM, 2007. ISBN 978-1-59593-882-4.

[43] Y. Yu, Y. Lin, Z. Hu, S. Hidaka, H. Kato, and L. Montrieux. Maintaining invariant traceability through bidirectional transformations. In M. Glinz, G. C. Murphy, and M. Pezzè, editors, *ICSE*, pages 540–550. IEEE, 2012. ISBN 978-1-4673-1067-3.

Proofs

The supplemnetary material contains the proofs of Theorems 1, 3 and 4, and Lemma 4 in this order.

A. Proof of Theorem 1

We will use the type system defined in Section 6 to prove the theorem. By using the type system, we can have the following lemma.

Lemma 10. *Suppose $\Delta, x :: \mathcal{G} \vdash e :: \mathcal{G}$. Then, if $e \Rightarrow^* E[x]$ for some call-by-name evaluation context E (i.e., $E ::= \Box \mid E\ e \mid (\mathsf{fold}\ n\ e)\ E$), E must be the hole.*

Proof. It is easy to see the type system has the preservation property, and thus $\Delta \vdash E[x] :: \mathcal{G}$. Only the typable context is $\Box$. $\Box$

To prove Theorem 1, we prove the following more general property; Theorem 1 is the special case where g below is a closed UnCAL expression.

Theorem 6. *Suppose $\Gamma \vdash g :: \mathsf{DB}^X_Y \rightsquigarrow e$. Let Δ be a typing environment satisfying that $\Delta \vdash e :: \mathcal{G}^{|Y|} \to \mathcal{G}^{|X|}$ holds and that, for any x, $\Gamma(x) = \mathsf{DB}^{X'}_{Y'}$ implies $\Delta(x) = (\mathcal{G}')^{|Y'|} \to (\mathcal{G}')^{|X'|}$. Let θ and η be substitutions that have the same domain and satisfy that, for any x in the domain, $[\![\theta(x)]\!] \approx \eta(x)$, $\vdash \theta(x) :: \Gamma(x)$ and $\vdash \eta(x) :: \Delta(x)$ hold. Then, we have $[\![g\theta]\!] \approx e\eta$.*

Here, we write $g\theta$/$e\eta$ for the UnCAL/functional expression obtained from g/e by replacing each variable x with $\theta(x)$/$\eta(x)$.

We prove the theorem by the induction on the derivation tree of a conversion judgment $\Gamma \vdash g :: \mathsf{DB}^X_Y \rightsquigarrow e$. We only show the nontrivial cases: C-SUBST, C-CYC and C-REC. For simplicity, we restrict ourselves to the case where $X = \{\&\}$; the proof for the general case can be obtained straightforwardly from the discussion below. In the discussion below, we abuse the notation to apply graph constructors directly to UnCAL graphs, or (V, E, I, O)-tuples, according to the semantics shown in Section 3.

Case: C-SUBST We will prove

$$[\![(g_1 \mathbin{@} g_2)\theta]\!] \approx \lambda y. e_1 (e_2 y)\eta$$

where $\Gamma \vdash g_1 :: \mathsf{DB}^{\&}_Y \rightsquigarrow e_1$ and $\Gamma \vdash g_2 :: \mathsf{DB}^Y_Z \rightsquigarrow e_2$. Let e'_1 and e'_2 be $e_1\eta$ and $e_2\eta$. To prove this, it suffices to prove

$$[\![(g_1 \mathbin{@} g_2)\theta]\!] \mathbin{@} G_3 \sim (e'_1\ (e'_2\ e_3))$$

for any G_3 and e_3 satisfying $G_3 \sim e_3$.

Let G, G_1, and G_2 be graphs $[\![(g_1 \mathbin{@} g_2)\theta]\!] \mathbin{@} G_3$, $[\![g_1\theta]\!]$, and $[\![g_2\theta]\!]$, respectively. Then, G can be written as $G = (G_1 \mathbin{@} G_2) \mathbin{@} G_3$. Since @ is associative, we can rewrite it to $G = G_1 \mathbin{@} (G_2 \mathbin{@} G_3)$. From the induction hypothesis, we have $G_2 \mathbin{@} G_3 \sim (e'_2\ e_3)$ and thus $G_1 \mathbin{@} (G_2 \mathbin{@} G_3) \sim e'_1\ (e'_2\ e_3)$.

Case: C-CYC We will prove

$$[\![\mathbf{cycle}(g)\theta]\!] \approx \lambda \overline{Y}. \mathsf{fix}_{\mathbf{G}}(\lambda \overline{X}. e\ \overline{X \cup Y})\eta$$

where $\theta \vdash g :: \mathsf{DB}^X_{X \uplus Y} \rightsquigarrow e$. As wrote above, we only consider the case where $X = \{\&\}$. To prove this, it suffices to show

$$\begin{aligned}&[\![\mathbf{cycle}(g)\theta]\!] \mathbin{@} G_2\\ &\sim \mathsf{fix}_{\mathbf{G}}(\lambda x. e\ (\overline{\{x\} \cup Y}[\pi_1 e_2/y_1, \dots, \pi_m e_2/y_m])\eta)\end{aligned}$$

for any (G_2, e_2) such that $G_2 \sim e_2$. Here x is the variable corresponds to the marker &. We write e' for $e\eta$, and $e'_1, \dots, e'_n$ for $e_1\eta, \dots, e_n\eta$.

Let G be $([\![\mathbf{cycle}(g)\theta]\!] \mathbin{@} G_2)$, and G_1 be $[\![g\theta]\!]$. Then, G and $G_1 \mathbin{@} G_2$ only differ in the nodes that have the output marker $\&x$. That is, a node that has the output marker $\&x$ in $G_1 \mathbin{@} G_2$ has an ε-edge to the node indicated by the input marker $\&x$ in G, assuming that we have used the same set of nodes V for G and $G_1 \mathbin{@} G_2$. We write $\lambda x. C_0[x]$ for $\lambda x. e'\ (\overline{\{x\} \cup Y}[\pi_1 e'_2/y_1, \dots, \pi_m e'_2/y_m])$. Then, we have $G_1 \mathbin{@} G_2 \approx \lambda x. C_0[x]$ from the induction hypothesis.

Let H be a graph that consists only of one edge with special label $\mathtt{0}$ that does not occur in any other places, and v_H be its root. It is easy to show that $H \approx \mathtt{0} : \Omega$ where Ω is a free variable. Then, we discuss the evaluation of $\mathsf{fix}_{\mathbf{G}}(\lambda x. C_0[x])$ and $C_0[\mathtt{0} : \Omega]$. Since we have $G_1 \mathbin{@} G_2 \mathbin{@} H \sim C_0[\mathtt{0} : \Omega]$ from $G_1 \mathbin{@} G_2 \approx \lambda x. C_0[x]$, we have a bisimulation $\mathcal{X}$ to prove this $\sim$. Let G' be $(G_1 \mathbin{@} G_2 \mathbin{@} H)$, and e_0 be $\mathsf{fix}_{\mathbf{G}}(\lambda x. C_0[x])$. We define $\mathcal{X}'$ as

$$\begin{array}{ll}(v, C[e_0]) \in \mathcal{X}' & \text{if } (v, C[\mathtt{0} : \Omega]) \in \mathcal{X}, v \in V\\ (v_0, e_0) \in \mathcal{X}' & \text{if } v_0 \text{ is the root of } G\end{array}$$

and will show that $\mathcal{X}'$ is a bisimulation between G and e_0.

Suppose that we have $(v, e'') \in \mathcal{X}'$ and $v \to^{\varepsilon} \cdots \to^{\varepsilon} \to^{\mathtt{a}} v'$. There are two cases: (1) all edges are also in G', (2) some edges are not in G'. For Case (1), from the definition of $\mathcal{X}'$, there are two possibilities: (1-i) $v = v_0$ and $e'' = e_0$, and (1-ii) $e'' = C[e_0]$ where $(v, C[\mathtt{0} : \Omega]) \in \mathcal{X}$. We can safely ignore Case (1-i) because we have $e_0 \to^* C[e_0]$ and $(v_0, C[\mathtt{0} : \Omega]) \in \mathcal{X}$; the proof is covered by the second case. For Case (1-ii), where $e'' = C[e_0]$ with $(v, C[\mathtt{0} : \Omega]) \in \mathcal{X}$, there is some C' such that $(v', C'[\mathtt{0} : \Omega]) \in \mathcal{X}$ and $C[\mathtt{0} : \Omega] \Rightarrow^* \mathtt{a} : C'[\mathtt{0} : \Omega]$ from Lemma 10 (note that we have $x :: \mathcal{G} \vdash C[x] :: \mathcal{G}$, which can be shown by the preservation property and $x :: \mathcal{G} \vdash C_0[x] :: \mathcal{G}$). Then, we have $C[e_0] \Rightarrow^* \mathtt{a} : C'[e_0]$ and, by definition, $(v', C'[e_0]) \in \mathcal{X}$. For Case (2), it is suffice to consider the case where there is one edge $u \to^{\varepsilon} u'$ that is not in G'. Then, u' must be v_0 from the evaluation of **cycle** and $u \to^{\mathtt{0}} v_H$ in G'. Thus, there must be the reduction $C[\mathtt{0} : \Omega] \Rightarrow^* C'[\mathtt{0} : \Omega] \Rightarrow^* \mathtt{0} : \Omega$ from the bisimilarity between v and e'' with respect to $\mathcal{X}$. Then, we must have the sequence $C[e_0] \Rightarrow^* C'[e_0] \Rightarrow^* e_0$. We have $(v_0, e_0) \in \mathcal{X}$ and $e_0 \to^* C_0[e_0]$. Since the node v_0 is also the root of the graph G', we have $(v_0, C_0[\mathtt{0} : \Omega]) \in \mathcal{X}$ and thus $(v_0, C_0[e_0]) \in \mathcal{X}'$. The rest of the proof is similar to Case (1).

Suppose that we have $(v, e'') \in \mathcal{X}'$ and $e'' \Rightarrow^* \mathtt{a} : e'''$. Similarly, there are three cases: (1) there is a corresponding evaluation sequence $C_1[\mathtt{0} : \Omega] \Rightarrow C_2[\mathtt{0} : \Omega] \Rightarrow \dots \Rightarrow \mathtt{a} : C_n[\mathtt{0} : \Omega]$ such that $e'' = C_1[e_0]$ and $e''' = C_n[e_0]$, (2) $v = v_0$ and $e'' = e_0$, and (3) $e'' = C[e_0]$ but there is no corresponding evaluation sequence as Case (1). The discussion for Case (1) is a counterpart of Case (1) in the previous paragraph, so we omit the discussion. Case (2) can be proved similarly to Case (1) because e_0 has the reduction that $e_0 \to^* C_0[e_0]$ (recall that $\to$ is the call-by-name reduction) and we have $(v_0, C_0[e_0]) \in \mathcal{X}'$ from $(v_0, C_0[\mathtt{0} : \Omega]) \in \mathcal{X}$. For Case (3), since $C[e_0]$ and $C[\mathtt{0} : \Omega]$ must have the corresponding reduction unless $C = \Box$, which is obtained from Lemma 10, the evaluation sequence must has the form of $e'' \Rightarrow^* e_0 \Rightarrow^* C[e_0] \Rightarrow^* \mathtt{a} : e'''$, while the corresponding evaluation sequence is $C[\mathtt{0} : \Omega] \Rightarrow^* \mathtt{0} : \Omega$. Then, we have $v \to^{\varepsilon} \cdots \to^{\varepsilon} v_H \to^{\mathtt{0}} v'$ in G'. Note that v_H is the only the node having an $\mathtt{0}$-labeled edge. According to the semantics of UnCAL, we can say that if there is a ε-path from v to v_H in G', there also is an ε-path from v to v_0 in G. Thus, we have an ε-path from v to v_0. Since we have $(v_0, C[e_0]) \in \mathcal{X}'$, the rest of the proof can be done by iterating this discussion and Case (1) because the evaluation sequence $e'' \Rightarrow^* \mathtt{a} : e'''$ is finite.

Case: C-REC We will prove

$$\begin{aligned}&[\![\mathbf{srec}(\lambda(l,t).g_1)(g_2)\theta]\!]\\ &\approx \lambda y. \mathsf{para}\ |Z|\ (\lambda l. \lambda t. (\lambda t. e_1)\ (\lambda\_. t))\ (e_2\ ())\eta\end{aligned}$$

where $\Gamma, l :: \mathsf{L}, t :: \mathsf{DB}^{\{\&\}}_{\emptyset} \vdash g_1 :: \mathsf{DB}^Z_Z \rightsquigarrow e_1$ and $\Gamma \vdash g_2 :: \mathsf{DB}^{\&}_{\emptyset} \rightsquigarrow e_2$. For simplicity, we assume that $Z = \{\&z\}$; The general-case proof is a straightforward extension of this case. Let g_1', g_2', e_1' and e_2' be expressions $g_1\theta$, $g_2\theta$, $e_1\eta$ and $e_2\eta$, respectively. It suffices to show

$$[\![\mathbf{srec}(\lambda(l,t).g_1')(g_2')]\!] \sim \mathsf{para}\ 1\ (\lambda l.\lambda t'.(\lambda t.e_1')\ (\lambda().t'))\ (e_2'\ ())$$

to prove the above. For simplicity, we directly handle para instead of fold. We write P for the expression $\mathsf{para}\,1(\lambda l.\lambda t'.(\lambda t.e_1')\ (\lambda().t'))$.

Let G_0 and G_2 be graphs $[\![\mathbf{srec}(\lambda(l,t).g_1')(g_2')]\!]$ and $[\![g_2']\!]$, respectively. From the induction hypothesis, we have $G_2 \sim e_2'\ ()$; we write $\mathcal{X}_2$ for the corresponding bisimulation. For each edge ζ in G_2, let H_ζ be the graph that consists only of one edge with the special label $\mathtt{0}_\zeta$, and r_ζ be its root. From the induction hypothesis, we have $[\![g_1[\mathtt{a}/l, G/t]]\!]\ @\ (\&z \triangleright H_\zeta) \sim e_1'[\mathtt{a}/l, e/t]\ (\mathtt{0}_\zeta : \Omega)$ for any G, e with $G \approx e$, where $\mathtt{a}$ is the label of the edge ζ and Ω is a free variable. We write $\mathcal{X}_{\zeta,G,e}$ for the corresponding bisimulation relation. We write $G_\zeta = (V_\zeta, E_\zeta, \_, \_)$ for the subgraph in G_0 obtained from an edge ζ in G_2, and u_v for the node in G_0 obtained from the node v in G_2, according to the semantics of $\mathbf{srec}$ in Section 3.

We define a relation $\mathcal{X}'$ as:

- $(v, C[P\ e_2'']) \in \mathcal{X}'$ if $v \in V_\zeta$, $(v, C[\mathtt{0}_\zeta : \Omega]) \in \mathcal{X}_{\zeta,G,e}$ and $(u, e_2'') \in \mathcal{X}_2$ where $(\_, \_, u) = \zeta$.
- $(v, P\ e) \in \mathcal{X}'$ if $v = u_{v'}$ and $(v', e) \in \mathcal{X}_2$

Then, we will prove that the $\mathcal{X}'$ is a bisimulation to prove $G_0 \sim P\ (e_2\ ())$.

Suppose that we have $(v, e) \in \mathcal{X}'$ and $v \to^\varepsilon \cdots \to^\varepsilon \to^{\mathtt{a}} v'$. We have the two possibilities according to the two branches in the definition of $\mathcal{X}'$. We only consider the former case because, for the latter case, we have v'' and e'' such that we have $v \to^\varepsilon v''$ in the sequence and $e \to^* e''$, satisfying the former condition. The proof is straightforward if the path from v to v' contains only the nodes in V_ζ for some ζ. Otherwise, there must be the node v'', u_{v_2}, v''' in the sequence such that $v \to^\varepsilon \cdots \to^\varepsilon v'' \to^\varepsilon u_{v_2} \to^\varepsilon v''' \to^\varepsilon \ldots$ and $v'' \in V_\zeta$ and $v''' \in V_{\zeta'}$. It suffices to consider the case where there is one such nodes because other cases can be obtained by the induction on the length of the path. In such a case, since e has the form $C[P\ e_2'']$ where $(v, C[\mathtt{0}_\zeta : \Omega]) \in \mathcal{X}_{\zeta,G',e'}$ for some G' and e'. Since $v'' \to^{\mathtt{0}_\zeta} r_\zeta$, there is a reduction sequence $C[\mathtt{0}_\zeta : \Omega] \Rightarrow^* \mathtt{0}_\zeta : \Omega$. Thus, there must also be a reduction sequence $C[P\ e_2''] \Rightarrow^* P\ e_2''$. Since $\zeta = (\_, \_, v_2)$ from the semantics of $\mathbf{srec}$ and $(v_2, e_2'') \in \mathcal{X}_2$, we have $(u_{v_2}, P\ e'') \in \mathcal{X}'$. Then, the rest of the discussion is the similar to the latter case.

We omit the converse direction because the idea of the proof is similar to that for C-CYC. □

B. Proof of Theorem 3

The basic idea is to assign a natural number to an expression so that an expression that occurs as an argument of $\mathbf{srec}$ has a greater number than outer expressions. Formally, we prove the theorem by extending $\Gamma \vdash \mathsf{DB}^X_Y :: g \rightsquigarrow e$ to the 6-ary relation $\Gamma \vdash \mathsf{DB}^X_Y :: g \rightsquigarrow e :: \tau, \Delta$ that reads an UnCAL expression g that has type DB^X_Y under a typing environment Γ is converted to an expression e that has type τ under a typing environment Δ in the type system in Section 6.

Figure 9 shows the conversion rules. Here, $\Delta_1 \sqcup \Delta_2$ is defined as:

$$(\Delta_1 \sqcup \Delta_2)(x) = \begin{cases} \tau_1 \sqcup \tau_2 & (x \in \Delta_1 \wedge x \in \Delta_2) \\ \tau_1(x) & (x \in \Delta_1 \wedge x \notin \Delta_2) \\ \tau_2(x) & (x \notin \Delta_1 \wedge x \in \Delta_2) \\ \bot & \text{otherwise} \end{cases}$$

Here, $\tau_1 \sqcup \tau_2$ denotes the least upper bound of τ_1 and τ_2 with respect to the subtyping relation $\preceq$. We assume that Γ only contains graph variables of type $\mathsf{DB}^{\&}_{\emptyset}$; accordingly, Δ maps a variable x to type $() \to \bigcirc^n\mathsf{G}$ for some n. Note that this property is closed under $\sqcup$. The rules for graph constructors basically do not touch $\bigcirc$. The only rule that uses $\bigcirc$ is C-REC. The rule C-REC says that the argument of $\mathbf{srec}(\lambda(l,t).g_1)$ must be more traversal than its return type and t. It is worth noting that every τ of $\Gamma \vdash \mathsf{DB}^X_Y :: g \rightsquigarrow e :: \tau, \Delta$ in Figure 9 has the form of $(\bigcirc^n\mathsf{G})^{|Y|} \to (\bigcirc^n\mathsf{G})^{|Z|}$ for some n.

Unlike the type system shown in Section 6, we consider tuples here. Accordingly, we extend the type to include (first-order) product-types T as

$$\tau ::= \cdots \mid \tau_1 \times \cdots \times \tau_n$$

and extend the typing rules for $\mathsf{fix}_{\mathbf{G}}$ and fold_e to return tuples of graphs as below.

$$\frac{\Gamma \vdash e :: (\mathcal{G}_1 \times \cdots \times \mathcal{G}_n) \to (\mathcal{G}_1 \times \cdots \times \mathcal{G}_n)}{\Gamma \vdash \mathsf{fix}_{\mathbf{G}}\ e :: (\mathcal{G}_1 \times \cdots \times \mathcal{G}_n)}\ \text{T-FIX}$$

$$\frac{\Gamma \vdash e :: \mathsf{L} \to (\mathcal{G}_1 \times \cdots \times \mathcal{G}_n) \to (\mathcal{G}_1 \times \cdots \times \mathcal{G}_n)}{\Gamma \vdash \mathsf{fold}_n e :: \bigcirc(\mathcal{G}_1 \sqcup \cdots \sqcup \mathcal{G}_n) \to (\mathcal{G}_1 \times \cdots \times \mathcal{G}_n)}\ \text{T-CATA}$$

We omit the type rules for the tuple construction and projections because they are standard ones. The typing rule T-CATA says that the argument must has more traversability than any component of return value. Thus, the derived operator $\mathsf{para}_n e$ must obey the following typing rule.

$$\frac{\Gamma \vdash e :: \mathsf{L} \to \mathcal{G} \to (\mathcal{G}_1 \times \cdots \times \mathcal{G}_n) \to (\mathcal{G}_1 \times \cdots \times \mathcal{G}_n)}{\Gamma \vdash \mathsf{para}_n e :: \bigcirc(\mathcal{G} \sqcup \mathcal{G}_1 \sqcup \cdots \sqcup \mathcal{G}_n) \to (\mathcal{G}_1 \times \cdots \times \mathcal{G}_n)}$$

Now, we can prove Theorem 3 by showing the following two properties:

- Suppose $\Gamma \vdash g :: \mathsf{DB}^X_Y \rightsquigarrow e :: \tau, \Delta$, then $\Delta \vdash e :: \tau$.
- Suppose $\Gamma \vdash g :: \mathsf{DB}^X_Y \rightsquigarrow e$, then $\Gamma \vdash g :: \mathsf{DB}^X_Y \rightsquigarrow e :: \tau, \Delta$ holds for some τ and Δ.

The former property is proved straightforwardly by the induction on the derivation tree. To prove the latter property, the point is to guarantee that the premise of the extended conversion rule for $\mathbf{srec}$ holds if the corresponding premise of the original conversion rule for $\mathbf{srec}$ holds. This is done by proving the following more general property.

Lemma 11. *Suppose* $\Gamma \vdash g :: \mathsf{DB}^X_Y \rightsquigarrow e$*, then, for any natural number n, there is* Δ *such that* $\Gamma \vdash g :: \mathsf{DB}^X_Y \rightsquigarrow e :: (\bigcirc^n\mathsf{G})^{|Y|} \to (\bigcirc^n\mathsf{G})^{|X|}, \Delta$ *holds.* □

Unlike the original property, the induction hypothesis used in the inductive proof of this lemma becomes strong enough to prove the T-CATA-case.

C. Proof of Theorem 4

First, we define the logical relation $\mathcal{B}$ as follows.

Definition 3. The binary relation on expressions $\mathcal{B}$ is defined as follows.

- $(e_1, e_2) \in\mathcal{B}_{\mathsf{L}}$ iff $e_1 \to^* \mathtt{a}$ and $e_2 \to^* \mathtt{a}$ for some $\mathtt{a}$,
- $(e_1, e_2) \in\mathcal{B}_{\mathcal{G}}$ iff $G_{e_1} \sim G_{e_2}$,
- $(f_1, f_2) \in\mathcal{B}_{\tau_1\to\tau_2}$ iff $(f_1 e_1, f_2 e_2) \in\mathcal{B}_{\tau_2}$ for any $(e_1, e_2) \in\mathcal{B}_{\tau_2}$

Intuitively, $\mathcal{B}_\tau$ extends the graph bisimilarity $\sim$ to higher-order expressions.

Then, we prove the following lemma, which is a generalization of Theorem 4.

Lemma 12. *Suppose* $\Gamma \vdash e :: \tau$*. We have* $(e\theta_1, e\theta_2) \in\mathcal{B}_\tau$ *where* θ_1 *and* θ_2 *are substitutions satisfying* $(\theta_1(x), \theta_2(x)) \in\mathcal{B}_{\Gamma(x)}$ *for all* $x \in \mathsf{dom}(\Gamma)$.

$$\frac{}{\Gamma \vdash \{\} :: \mathsf{DB}^{\{\&\}}_Y \rightsquigarrow \bullet :: () \to \bigcirc^n \mathsf{G}, \emptyset}\ \textsc{C-Single}$$

$$\frac{\Gamma \vdash g :: \mathsf{DB}^{\{\&\}}_Y \rightsquigarrow e :: (\bigcirc^n \mathsf{G})^{|Y|} \to \bigcirc^n \mathsf{G}, \Delta}{\Gamma \vdash \{l : g\} :: \mathsf{DB}^{\{\&\}}_Y \rightsquigarrow (\lambda y.l : e) :: (\bigcirc^n \mathsf{G})^{|Y|} \to \bigcirc^n \mathsf{G}, \Delta}\ \textsc{C-Edge}$$

$$\frac{\left\{\Gamma \vdash g_i :: \mathsf{DB}^{\&}_Y \rightsquigarrow e_i :: (\bigcirc^n \mathsf{G})^{|Y|} \to \bigcirc^n \mathsf{G}, \Delta_i\right\}_{i=1,2}}{\Gamma \vdash g_1 \cup g_2 :: \mathsf{DB}^{\&}_Y \rightsquigarrow (\lambda y.(e_1 y) \cup (e_2 y)) :: (\bigcirc^n \mathsf{G})^{|Y|} \to \bigcirc^n \mathsf{G}, \Delta_1 \sqcup \Delta_2}\ \textsc{C-Uni}$$

$$\frac{\Gamma \vdash g :: \mathsf{DB}^{\{\&\}}_Y \rightsquigarrow e :: (\bigcirc^n \mathsf{G})^{|Y|} \to \bigcirc^n \mathsf{G}, \Delta}{\Gamma \vdash \&x := g :: \mathsf{DB}^{\{\&x\}}_Y \rightsquigarrow e :: (\bigcirc^n \mathsf{G})^{|Y|} \to \bigcirc^n \mathsf{G}, \Delta}\ \textsc{C-Root}$$

$$\frac{y = y_i \text{ of } (y_1, \ldots, y_n) = \overline{Y}}{\Gamma \vdash \&y :: \mathsf{DB}^{\{\&\}}_Y \rightsquigarrow \lambda y.\pi_i y :: (\bigcirc^n \mathsf{G})^{|Y|} \to \bigcirc^n \mathsf{G}, \emptyset}\ \textsc{C-Hole}$$

$$\frac{}{\Gamma \vdash () :: \mathsf{DB}^{\emptyset}_Y \rightsquigarrow \lambda y.() :: (\bigcirc^n \mathsf{G})^{|Y|} \to (), \emptyset}\ \textsc{C-Emp}$$

$$\frac{\{\Gamma \vdash g_i :: \mathsf{DB}^{X_i}_Y \rightsquigarrow e_i :: (\bigcirc^n \mathsf{G})^{|Y|} \to (\bigcirc^n \mathsf{G})^{|X_i|}, \Delta_i\}_{i=1,2} \quad p \equiv \lambda\overline{X_1}.\lambda\overline{X_2}.\overline{X_1 \cup X_2}}{\Gamma \vdash g_1 \oplus g_2 :: \mathsf{DB}^{X_1 \uplus X_2}_Y \rightsquigarrow \lambda y.p\,(e_1\ y)\ (e_2\ y) :: (\bigcirc^n \mathsf{G})^{|Y|} \to (\bigcirc^n \mathsf{G})^{|X_1 \cup X_2|}, \Delta_1 \sqcup \Delta_2}\ \textsc{C-RU}$$

$$\frac{\Gamma \vdash g_1 :: \mathsf{DB}^{X}_Y \rightsquigarrow e_1 :: (\bigcirc^n \mathsf{G})^{|Y|} \to (\bigcirc^n \mathsf{G})^{|X|}, \Delta_1 \quad \Gamma \vdash g_2 :: \mathsf{DB}^{Y}_Z \rightsquigarrow e_2 :: (\bigcirc^n \mathsf{G})^{|Z|} \to (\bigcirc^n \mathsf{G})^{|Y|}, \Delta_2}{\Gamma \vdash g_1 \mathbin{@} g_2 :: \mathsf{DB}^{X}_Z \rightsquigarrow \lambda y.e_1\ (e_2\ y)(\bigcirc^n \mathsf{G})^{|Z|} \to (\bigcirc^n \mathsf{G})^{|X|}\Delta_1 \sqcup \Delta_2}\ \textsc{C-Subst}$$

$$\frac{\Gamma \vdash g :: \mathsf{DB}^{X}_{X \uplus Y} \rightsquigarrow e :: (\bigcirc^n \mathsf{G})^{|X \uplus Y|} \to (\bigcirc^n \mathsf{G})^{|X|}, \Delta}{\Gamma \vdash \mathbf{cycle}(g) :: \mathsf{DB}^{X}_Y \rightsquigarrow \lambda\overline{Y}.\,\mathsf{fix}_{\mathbf{G}}(\overline{X}.e\ \overline{X \cup Y}) :: (\bigcirc^n \mathsf{G})^{|Y|} \to (\bigcirc^n \mathsf{G})^{|X|}, \Delta}\ \textsc{C-Cyc}$$

$$\frac{}{\Gamma \vdash y :: \Gamma(y) \rightsquigarrow y :: () \to \bigcirc^n \mathsf{G}, \{y :: \bigcirc^n \mathsf{G}\}}\ \textsc{C-Var}$$

$$\frac{\Gamma \vdash g :: \mathsf{DB}^{X}_Y \rightsquigarrow e :: (\bigcirc^n \mathsf{G})^{|Y|} \to (\bigcirc^n \mathsf{G})^{|X|}, \Delta \quad Y \subseteq Y'}{\Gamma \vdash g :: \mathsf{DB}^{X}_{Y'} \rightsquigarrow \lambda\overline{Y'}.e\ \overline{Y} :: (\bigcirc^n \mathsf{G})^{|Y'|} \to (\bigcirc^n \mathsf{G})^{|X|}, \Delta}\ \textsc{C-Sub}$$

$$\frac{\begin{array}{l}\Gamma, l :: \mathsf{L}, t :: \mathsf{DB}^{\{\&\}}_{\emptyset} \vdash g_1 :: \mathsf{DB}^{Z}_Z \rightsquigarrow e_1 :: (\bigcirc^n \mathsf{G})^{|Z|} \to (\bigcirc^n \mathsf{G})^{|Z|}, \Delta_1 \quad \Delta_1(t) = () \to (\bigcirc^m \mathsf{G}) \\ \Gamma \vdash g_2 :: \mathsf{DB}^{\{\&\}}_{\emptyset} \rightsquigarrow e_2 :: () \to (\bigcirc^{\max\{m,n\}+1}\mathsf{G}), \Delta_2\end{array}}{\Gamma \vdash \mathbf{srec}(\lambda(l,t).g_1)(g_2) :: \mathsf{DB}^{Z}_{\emptyset} \rightsquigarrow \lambda y.\mathsf{para}_{|Z|}\ (\lambda l.\lambda t'.(\lambda t.e_1)\ (\lambda().t'))\ (e_2\ ()) :: () \to (\bigcirc^n \mathsf{G})^{|Z|}, \Delta_1 \sqcup \Delta_2}\ \textsc{C-Rec}$$

Figure 9. Conversion rules of UnCAL expressions

Proof. We prove the statement by induction on the typing derivation. We shall only show the nontrivial cases, T-FIX and T-CATA. Note that we ignore memos in this section because we consider call-by-name reduction instead of the abstract machine in Section 5.

Case: T-FIX. We will prove $\mathsf{fix}_{\mathbf{G}}\, e\theta_1 \sim \mathsf{fix}_{\mathbf{G}}\, e\theta_2$. Let us write e_1 for $e\theta_1$ and e_2 for $e\theta_2$. Then we have $(e_1, e_2) \in \mathcal{B}_{\mathcal{G} \to \mathcal{G}}$ from the induction hypothesis. Thus, we have $(e_1\ \Omega) \sim (e_2\ \Omega)$ where Ω is a fresh free variable. Let us write $\mathcal{X}'$ for the bisimulation relation to prove $(e_1\ \Omega) \sim (e_2\ \Omega)$.

Then, we define $\mathcal{X}$ as follows.

$$\begin{array}{l}(\mathsf{fix}_{\mathbf{G}}\, e_1, \mathsf{fix}_{\mathbf{G}}\, e_2) \in \mathcal{X} \\ (C_1[\mathsf{fix}_{\mathbf{G}}\, e_1], C_2[\mathsf{fix}_{\mathbf{G}}\, e_2]) \in \mathcal{X} \quad \text{if} \quad (C_1[\Omega], C_2[\Omega]) \in \mathcal{X}'\end{array}$$

We will prove that $\mathcal{X}$ is the bisimulation relation between $\mathsf{fix}_{\mathbf{G}}\, e_1$ and $\mathsf{fix}_{\mathbf{G}}\, e_2$.

Assume that $\mathsf{fix}_{\mathbf{G}}\, e_1 \Rightarrow^* \mathtt{a} : e'$. Let us focus on how $\mathsf{fix}_{\mathbf{G}}\, e_1$ is reduced. There are only the following two possibilities from Lemma 10.

- $\mathsf{fix}_{\mathbf{G}}\, e_1 \Rightarrow e_1(\mathsf{fix}_{\mathbf{G}}\, e_1) \Rightarrow^* C_1[\mathsf{fix}_{\mathbf{G}}\, e_1] \Rightarrow^* \mathtt{a} : C_1'[\mathsf{fix}_{\mathbf{G}}\, e_1]$ where $e_1\Omega \Rightarrow^* C_1[\Omega] \Rightarrow^* \mathtt{a} : C_1'[\Omega]$, or
- $\mathsf{fix}_{\mathbf{G}}\, e_1 \Rightarrow e_1(\mathsf{fix}_{\mathbf{G}}\, e_1) \Rightarrow^* C_1[\mathsf{fix}_{\mathbf{G}}\, e_1] \Rightarrow^* \mathsf{fix}_{\mathbf{G}}\, e_1$ where $e_1\Omega \Rightarrow^* C_1[\Omega] \Rightarrow^* \Omega$.

Then, e' must be the form of $C_1'[\mathsf{fix}_{\mathbf{G}}\, e_1]$ from the above discussion. Then, there must be a corresponding sequence from $\mathsf{fix}_{\mathbf{G}}\, e_2$ such that $\mathsf{fix}_{\mathbf{G}}\, e_2 \Rightarrow e_2(\mathsf{fix}_{\mathbf{G}}\, e_2) \Rightarrow^* C_2[\mathsf{fix}_{\mathbf{G}}\, e_2] \Rightarrow^* \mathtt{a} : C_2'[\mathsf{fix}_{\mathbf{G}}\, e_2]$ from the induction hypothesis. As a result, from the definition, we obtain $(C_1'[\mathsf{fix}_{\mathbf{G}}\, e_1], C_2'[\mathsf{fix}_{\mathbf{G}}\, e_2]) \in \mathcal{X}$.

Assume that $C_1[\mathsf{fix}_{\mathbf{G}}\, e_1] \Rightarrow^* \mathtt{a} : e'$ where $(C_1[\Omega], C_2[\Omega]) \in \mathcal{X}'$. Let us consider how $C_1[\mathsf{fix}_{\mathbf{G}}\, e_1]$ is evaluated. There are only the two possibilities from Lemma 10:

- $C_1[\mathsf{fix}_{\mathbf{G}}\, e_1] \Rightarrow^* \mathtt{a} : C_1'[\mathsf{fix}_{\mathbf{G}}\, e_1]$ where $C_1[\Omega] \Rightarrow^* \mathtt{a} : C_1'[\Omega]$, or
- $C_1[\mathsf{fix}_{\mathbf{G}}\, e_1] \Rightarrow^* \mathsf{fix}_{\mathbf{G}}\, e_1$ where $C_1[\Omega] \Rightarrow^* \Omega$.

For the former case, there must be a corresponding sequence from $C_2[\mathsf{fix}_{\mathbf{G}}\, e_2]$ such that $C_2[\mathsf{fix}_{\mathbf{G}}\, e_2] \Rightarrow^* \mathtt{a} : C_2'[\mathsf{fix}_{\mathbf{G}}\, e_2]$ where $C_2[\Omega] \Rightarrow^* \mathtt{a} : C_2'[\Omega]$. From the definition, we have $(C_1'[\mathsf{fix}_{\mathbf{G}}\, e_1], C_2'[\mathsf{fix}_{\mathbf{G}}\, e_2]) \in \mathcal{X}$. For the latter case, we repeat the discussion in the previous paragraph.

Case: T-CATA. We will prove $\mathsf{fold}(f\theta_1)\ e_1 \sim \mathsf{fold}(f\theta_2)\ e_2$ for any e_1 and e_2 such that $e_1 \sim e_2$. Let us write f_1 and f_2 for $f\theta_1$ and $f\theta_2$, respectively. From the induction hypothesis, we have $f_1\ \mathtt{a}\ \Omega \sim f_2\ \mathtt{a}\ \Omega$ for any $\mathtt{a}$ where Ω is a fresh free variable. Let us write $\mathcal{X}_{\mathtt{a}}'$ is the bisimulation relation to prove $f_1\ \mathtt{a}\ \Omega \sim f_2\ \mathtt{a}\ \Omega$, and $\mathcal{X}''$ is that to prove $e_1 \sim e_2$.

Then, we construct $\mathcal{X}$ as follows.

$$(\mathsf{fold}\, f_1\ e_1', \mathsf{fold}\, f_2\ e_2') \in \mathcal{X} \quad \text{if} \quad (e_1', e_2') \in \mathcal{X}''$$
$$(C_1[\mathsf{fold}\, f_1\ e_1'], C_2[\mathsf{fold}\, f_2\ e_2']) \in \mathcal{X}$$
$$\text{if} \quad (C_1[\Omega], C_2[\Omega]) \in \mathcal{X}'_{\mathtt{a}} \text{ and } (e_1', e_2') \in \mathcal{X}'' \text{ for some } \mathtt{a}$$

We will prove that $\mathcal{X}$ is the bisimulation relation between $\mathsf{fold}\, f_1 e_1$ and $\mathsf{fold}\, f_2 e_2$.

Assume that $\mathsf{fold}\, f_1 e_1' \Rightarrow^* \mathtt{a} : e''$. Let us consider how $\mathsf{fold}\, f_1 e_1'$ is reduced. From Lemma 10, there are only the two possibilities:

- $\mathsf{fold}\, f_1\ e_1' \Rightarrow^* \mathsf{fold}\, f_1\ (\mathtt{c} : e_1'') \Rightarrow^* f_1\ \mathtt{a}\ (\mathsf{fold}\, f_1 e_1'') \Rightarrow^* C_1[\mathsf{fold}\, f_1 e_1''] \Rightarrow^* \mathtt{b} : C_1'[\mathsf{fold}\, f_1 e_1'']$ where $e_1' \Rightarrow \mathtt{a} : e_1''$ and $f_1\ \mathtt{a}\, \Omega \Rightarrow^* C_1[\Omega] \Rightarrow^* \mathtt{b} : C_1'[\Omega]$, or
- $\mathsf{fold}\, f_1\ e_1' \Rightarrow^* \mathsf{fold}\, f_1\ (\mathtt{c} : e_1'') \Rightarrow^* f_1\ \mathtt{a}\ (\mathsf{fold}\, f_1 e_1'') \Rightarrow^* C_1[\mathsf{fold}\, f_1 e_1''] \Rightarrow^* \mathsf{fold}\, f_1 e_1''$ where $e_1' \Rightarrow \mathtt{a} : e_1''$ and $f_2\, \mathtt{a}\, \Omega \Rightarrow^* C_1[\Omega] \Rightarrow^* \Omega$.

In either case, $\mathsf{fold}\, f_2\ e_2'$ has the corresponding reduction from the induction hypothesis. Thus, it must be the case that $e'' = C_1[\mathsf{fold}\, f_1 e_1'']$ and $\mathsf{fold}\, f_2\ e_2'$ has the corresponding reduction $\mathsf{fold}\, f_2\ e_2' \Rightarrow^* C_2[\mathsf{fold}\, f_1 e_2''$ satisfying $(e_1'', e_2'') \in \mathcal{X}''$ and $(C_1[\Omega], C_2[\Omega]) \in \mathcal{X}'_{\mathtt{d}}$ for some $\mathtt{d}$. Then, from the definition, we obtain $(C_1[\mathsf{fold}\, f_1 e_1''], C_2[\mathsf{fold}\, f_1 e_2'']) \in \mathcal{X}$.

Assume that $C_1[\mathsf{fold}\, f_1\ e_1'] \Rightarrow^* \mathtt{a} : e''$. From Lemma 10, there are only two possibilities about how $C_1[\mathsf{fold}\, f_1\ e_1']$ is reduced:

- $C_1[\mathsf{fold}\, f_1 e_1''] \Rightarrow^* \mathtt{b} : C_1'[\mathsf{fold}\, f_1 e_1'']$ where $e_1' \Rightarrow \mathtt{a} : e_1''$ and $C_1[\Omega] \Rightarrow^* \mathtt{b} : C_1'[\Omega]$, or
- $C_1[\mathsf{fold}\, f_1 e_1''] \Rightarrow^* \mathsf{fold}\, f_1 e_1''$ where $e_1' \Rightarrow \mathtt{a} : e_1''$ and $C_1[\Omega] \Rightarrow^* \Omega$.

In either case, $C_2[\mathsf{fold}\, f_2 e_2'']$ has the corresponding reduction sequence from the induction hypothesis. For the former case, the proof is trivial. For the latter case, we repeat the discussion in the previous paragraph. $\square$

D. Proof of Lemma 4

Let c be a configuration $\langle \mathsf{elim}_M\ e \mid \mu \rangle$. We prove it by showing that, for any substitution σ_1 and σ_2, $c\sigma_1$ is simulated by $c\sigma_2$. Then, if one sequence terminate, the all the others also terminate.

Let us consider the evaluation of $c\sigma_1 = \langle \mathsf{elim}_M\ e\sigma_1 \mid \mu\sigma_1 \rangle$ and $c\sigma_2 = \langle \mathsf{elim}_M\ e\sigma_2 \mid \mu\sigma_2 \rangle$. From the type, we can easy to see that the evaluation sequences from the two configurations may differ only from the place where the configurations $\langle \mathsf{elim}_{M'}\ z_i\sigma_1 \mid \mu'\sigma_1 \rangle$ and $\langle \mathsf{elim}_{M'}\ z_i\sigma_2 \mid \mu'\sigma_2 \rangle$ are evaluated. Here, M' is an extension of M, i.e., $M(v) = x$ implies $M'(v) = x$. However, for the both cases the evaluation terminates because either $z_i\sigma_j \in \mathsf{dom}(M')$ or $z_i\sigma_j = \bullet$ holds for $j = 1, 2$. $\square$