

A Functional Reformulation of UnCAL Graph-Transformations

Or, Graph Transformation as Graph Reduction

Kazutaka Matsuda

Tohoku University, Japan
kztk@ecei.tohoku.ac.jp

Kazuyuki Asada

The University of Tokyo, Japan
asada@kb.is.s.u-tokyo.ac.jp

Abstract

This paper proposes FUnCAL, a functional redesign of the graph transformation language UnCAL. A large amount of graph-structured data are widely used, including biological database, XML with IDREFs, WWW, and UML diagrams in software engineering. UnCAL is a language designed for *graph transformations*, i.e., extracting a subpart of a graph data and converting it to a suitable form, as what XQuery does for XMLs. 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 reasoning as well as optimization. However, 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, following the observation that markers can be emulated by tuples and λ -abstractions, we transform UnCAL programs to a restricted class of usual (thus, marker-free) functional ones. By this translation, we can reason, analyze or optimize UnCAL programs as usual functional programs. Moreover, we introduce a type system for showing that a small modification to the usual lazy semantics is enough to run well-typed functional programs as *finite*-graph transformations in a *terminating* way.

Categories and Subject Descriptors D.1.1 [Programming Techniques]: Applicative (Functional) Programming; D.3.2 [Programming Languages]: Language Classifications—Applicative (functional) language; H.2.3 [Database Management]: Languages—Query languages

General Terms Languages

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

1. Introduction

A large amount of graph-structured data are widely used, including biological information, XML with IDREFs, WWW, UML diagrams in software engineering [20], and Object Exchange Model (OEM) for exchanging arbitrary database structures [42]. In such circumstances, several languages, such as UnQL/UnCAL [8], Lorel [1], and 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 can be seen as an infinite (regular) tree. Bisimulation-based graph data structure has the merit of efficiency on equivalence checking [12]. Moreover, with 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 [27] as well as optimization [8, 24].

However, despite 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, 24] is quite similar to the classic fold-fusion [36].

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, this paper proposes FUnCAL, a subset of a usual functional programming language, and gives a translation from UnCAL programs to FUnCAL 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, which prevents us from directly importing existing program-manipulation techniques, is *markers* that connect two graphs or construct cycles. There are two sorts 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.

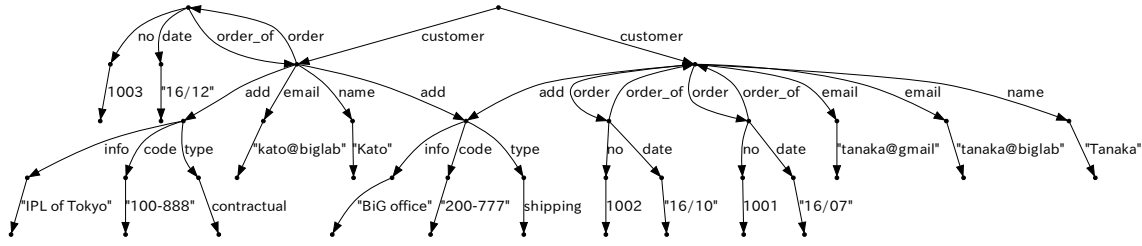


Figure 1. A graph data containing order information in customer-oriented representation.

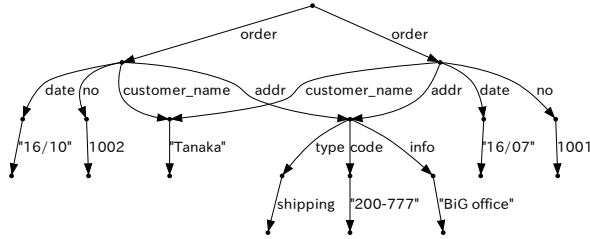


Figure 2. A graph data obtained from Figure 1 by extracting orders by “Tanaka” in order-oriented representation.

Let us review how markers are used in UnCAL. First, we explain UnCAL expressions that do not use any markers. Without markers, graphs in UnCAL are similar to records, as below.

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

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

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

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

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

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

$\{\text{name} : \text{Alice}, \text{friend} : \&x\}$
 $@ (\&x \triangleright \{\text{name} : \text{Bob}, \text{friend} : \&y\})$

we get the following graph.

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

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

$\text{cycle} \left(\&y \triangleright \left(\{\text{name} : \text{Alice}, \text{friend} : \&x\} \right. \right.$
 $\left. \left. @ (\&x \triangleright \{\text{name} : \text{Bob}, \text{friend} : \&y\}) \right) \right)$

The obtained graph represents that Alice and Bob are friends to each other.

All graph transformations in UnCAL are defined by srec , a structural recursion on graphs. With its bisimulation-respecting semantics, srec can be viewed as if it were defined recursively as:

$\text{srec}(e)(\{a_1 : G_1, \dots, a_n : G_n\}) =$
 $(e(a_1, G_1) @ \text{srec}(e)(G_1)) \cup \dots \cup (e(a_n, G_n) @ \text{srec}(e)(G_n))$

Here, \cup is the record concatenation; actually $\{x : s, y : t\}$ is a shorthand notation for $\{x : s\} \cup \{y : t\}$. The following example

returns people named Bob in db where db is a variable that stores a record of the form of $\{\text{person} : p_1, \dots, \text{person} : p_n\}$.

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

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

One might have noticed 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.

$\text{letrec } y = (\lambda x. \{\text{name} : \text{Alice}, \text{friend} : x\})$
 $\{\text{name} : \text{Both}, \text{friend} : y\} \text{ in } y$

The one also would have noticed that the behavior of srec could be expressed by a paramorphism [35] para that behaves like:

$\text{para } f \{a_1 : G_1, \dots, a_n : G_n\}$
 $= (f a_1 G_1 (\text{para } f G_1)) \cup \dots \cup (f a_n G_n (\text{para } f G_n))$

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

The straightforward translation is, however, unsatisfactory because the translation can map *terminating* UnCAL expressions to *nonterminating* ones. This is problematic if we apply optimization techniques such as fusion [36] 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 $\text{cycle}(\&x \triangleright \&x)$, which results in the singleton graph $\{\}$ in UnCAL, to $\text{letrec } x = x \text{ in } x$, which leads to an infinite loop in usual languages. Although the expression $\text{cycle}(\&x \triangleright \&x)$ 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 .

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

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 names. 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 $\text{cycle}(\&x \triangleright \{a : \&x\})$. The UnCAL expression

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

terminates and returns a singleton graph while the corresponding functional program

$\text{para } (\lambda_. \lambda_. \lambda r. r) (\text{letrec } x = \{a : x\} \text{ in } x)$

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

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). We name the target language of the transformation FUnCAL. 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, which is useful for importing verification techniques (Section 2.1). Next, to optimize UnCAL programs as functional ones, we give a semantics (Section 5) and a type system (Section 6) for FUnCAL, so that 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 typeability (Section 2.2). Note that our semantics itself is not new and nothing special; it is just the lazy semantics [38] with the black hole [2, 3, 38] and memoization. This helps us to implement the semantics easily, which runs faster than the existing implementation of UnCAL [26] (Section 2.3). Also, this enables us to bidirectionalize [33, 34] of UnCAL transformations (Section 2.4). In summary, in this paper:

- We formalize the transformation from UnCAL to functional ones (in FUnCAL) to support reasoning of UnCAL programs as functional ones (Section 4).
- We give the semantics and the type system of FUnCAL 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 by discussing 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 [33, 34, 46] 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 [30].

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. [27] 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 manipulate infinite trees instead of graphs. For example, thanks to our translation, we can use a verification method by Unno et al. [45], which is originally designed for tree transformations written in (higher-order) functional programs where the trees can be infinite, for graph transformations.

Although Inaba et al. [27]’s method is well tailored to UnCAL and thus the benefits are rather small for the “current” UnCAL, the advantage of our translation becomes clearer when we extend UnCAL. For example, if we extend UnCAL to include higher-order functions to improve the programmability as in [25], then Inaba et al. [27]’s method becomes no longer applicable. In contrast, the method by Unno et al. [45] is applicable for such extensions because it originally targets higher-order functional programs.

2.2 Optimization

Optimization is also important in graph transformation. There have been a few approaches for optimization of UnCAL programs [8, 24]. 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. Hidaka et al. [24], in addition to the basic idea, focus on manipulations of markers; for example, for $e @ (\&x \triangleright e')$, their transformation statically or dynamically computes the plugging-in operation by substituting $\&x$ in e by e' ; especially when e does not contain the output marker $\&x$, this replacement results in e .

The relationship between these UnCAL-specific techniques and 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 [36]. Since the expression $e @ (\&x \triangleright e')$ is converted to expression $(\lambda x. e) e'$ by our translation, we can regard Hidaka et al. [24]’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, such as supercompilation [43], short-cut fusion [19] and lightweight fusion [41]. Notice that we can freely optimize programs as long as the optimization keep typeability with respect to the type system in Section 6.

2.3 Implementation

Since our semantics (Section 5) is (a variant) of the usual lazy semantics, we can reuse existing implementation techniques. 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.

We implemented our semantics in Section 5 as an embedded DSL on Haskell;¹ inspired by [16], we represented laziness explicitly by a monad, and used a reference holding either a monadic computation or a pure value to represent a thunk and its evaluation result. An expected speed-up is reduction of 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; especially, any ε -edges [8] (see Section 3) are not used in our semantics.

We measured execution time of a few transformations and compared the execution time with GRoundTram 0.9.3 [23, 24, 26]², an UnCAL implementation using OCaml. GRoundTram implements

¹ <https://bitbucket.org/kztk/funscal>

² <http://www.biglab.org/download.html>

	$ V $	$ E $	Ours	GRoundTram (bulk)	— (rec)
Class2RDB	70	73	2.1 ms	39 s	58 ms
PIM2PSM	58	58	1.8 ms	6.7 s	12 ms
C20_Se1	42	45	1.4 ms	0.93 s	4.6 ms
a2d_xc S30k	30000	29999	0.62 s	1.3 s	1.7 s
a2d_xc M200	40000	80000	1.8 s	2.1 s	T/O
a2d_xc C200	201	40200	0.9 s	0.79 s	T/O

Table 1. Experimental results (running time is in CPU time, the timeout is 60 seconds).

two evaluation strategies: bulk semantics [8] that evaluates recursions as map-like operations on edges, and recursive semantics that uses memoized recursions to realize `srec`, both equipped with some optimizations [8, 24]. The experiments were held on MacOSX 10.11.6 over MacBook Pro 13-inch with 2.6 GHz Intel Core i5 CPU and 8 GB memory. We used GHC 8.0.1 (with LLVM 3.7) for Haskell and ocaml 4.03.0 for OCaml. Since GRoundTram is implemented as an interpreter, for fair comparison, we did not compile the tested programs and graphs; we used the interpreter `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 [24]) converts a platform independent model to a platform specific model, and `C20_Se1` (from [24]) converts a customer-order database from a customer-oriented representation to an order-oriented representation with some extraction, and `a2d_xc` (from [23]) renames `A` to `D` and contracts `C`. The program codes of `Class2RDB`, `PIM2PSM` and `C20_Se1` are mechanically generated; they are originally written in `UnQL+` [26] and converted to `UnCAL`. For `a2d_xc`, we used the three graphs as input: `S30k` is a 30000-long sequence of $\text{A} \rightarrow \text{A} \rightarrow \dots \rightarrow \text{A}$, `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`. Since our semantics is lazy, we used `isEmpty` (elim e) for deep evaluation (Section 5.3). For GRoundTram, we used options “`-oa -rw -cb`” for the former three experiments and “`-oa -rw -lu -le`” for the latter three. Additionally, for each experiment, we examined combination of the options `-as` and `-ht` and chose the fastest one.

Table 1 shows the experimental results. The results show that the bulk semantics (“bulk” in the table) in GRoundTram scales well for both node and edge size of a graph while its recursive semantics (“rec” in the table) scales well for the size of a query (`Class2RDB`, `PIM2PSM` and `C20_Se1` contain 57, 37 and 14 `srecs`, respectively). Though our semantics is also based on the recursive nature of `UnCAL`’s structural recursions, our implementation scaled enough well for both directions. This speed-up might be caused by the simplicity of the data structure used for a graph (thunks and constructors, as usual lazy evaluation). Especially, our semantics does not introduce any ε -edges intermediately, and thus, in general, the sizes of intermediate graphs are smaller than those in GRoundTram, even for the case where the whole intermediate graphs are demanded. However, unions represented as constructors in our system play similar roles to ε -edges, which explains our implementation does not scale well for non sparse graphs as `C200`.

We examined performance of application of an involved transformation to a non tiny graph. We artificially generated a graph that contains 1000 customers, each with 5 orders; the graph has 23006 nodes, and 45006 edges. Then, we applied `C20_Se1` to the graph. Our system took 3.1 CPU seconds, while GRoundTram’s bulk semantics (with options `-oa -rw -cb -as -ht`) took 30 CPU minutes and its recursive semantics caused the stack overflow. This slowdown of GRoundTram would come from the elimination of ε -edges produced by each `srec`. However, disabling ε -elimination,

GRoundTram ran out 8 GB memory after 30 minutes due to the size of intermediate graphs.

2.4 Bidirectionalization

A bidirectional transformation is a pair of a usual transformation of type $S \rightarrow V$ and a “backward” transformation of type $S \rightarrow V \rightarrow S$ that maps changes on the transformed data to the original data [17, 18, 23, 33, 34, 46]. 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 [47, 48]. Bidirectional semantics of `UnCAL` [23] has been studied for this application because usually those models are represented by graphs.

Our translation enables us to access bidirectionalization methods [33, 34, 46] studied for functional programs. In [33, 34], if a function f has polymorphic type

$$\forall \alpha. \forall \mu. \text{PackM } c \alpha \mu \Rightarrow T \alpha \rightarrow \mu (T' \alpha)$$

where T and T' are type constructors for container-like datatypes, namely instances of `Traversable`, and `PackM` $c \alpha \mu$ is a type class with the methods below³

$$\begin{aligned} \text{new} &:: c \rightarrow \alpha \\ \text{liftO} &:: \text{Eq } r \Rightarrow (c \rightarrow r) \rightarrow \alpha \rightarrow \mu r \\ \text{liftO2} &:: \text{Eq } r \Rightarrow (c \rightarrow c \rightarrow r) \rightarrow \alpha \rightarrow \alpha \rightarrow \mu r \end{aligned}$$

then we can derive a backward transformation of type $T c \rightarrow T' c \rightarrow T c$ corresponding to $f_{c,I} :: T c \rightarrow I (T' c)$ 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.

$$\text{eval} :: \text{PackM } L \alpha \mu \Rightarrow \text{Exp} \rightarrow \text{Env } \alpha \rightarrow \mu (\text{Graph } \alpha)$$

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 $l = a$ or $l = l'$, we use lifted comparisons `liftO` ($= a$) l or `liftO2` ($=$) $l l'$.

There already exists a bidirectional semantics [23] 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. [23]’s method becomes no longer applicable due to the higher-order constructs. In contrast, Matsuda and Wang [33, 34]’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 `UnQL+` [26], in which users can write queries like SQL; for example, the query that extracts a person named Bob shown in Section 1.1 can be written as follows.

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

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

Once a query is written in UnQL/UnQL⁺, it is converted to an UnCAL program and executed. This means, our translation is also beneficial to UnQL/UnQL⁺. Recently, UnCAL has been applied to bidirectional model-driven software development [23, 26, 48], where the structure of software is modeled as graphs in different levels of abstractions and their relationships are described by graph transformations.

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].

Formally, UnCAL graphs are defined as follows. 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, $E \subseteq V \times (L \cup \{\varepsilon\}) \times V$ 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 $v \xrightarrow{l} u$ 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 $\xrightarrow{\varepsilon}^*$ for the reflexive transitive closure of $\xrightarrow{\varepsilon}$. 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 $v_1 \xrightarrow{\varepsilon}^* w_1 \xrightarrow{a} u_1$ there is a path satisfying $v_2 \xrightarrow{\varepsilon}^* w_2 \xrightarrow{a} u_2$ and $(u_1, u_2) \in \mathcal{X}$, and for any path u_2 satisfying $v_2 \xrightarrow{\varepsilon}^* w_2 \xrightarrow{a} u_2$ there is a path satisfying $v_1 \xrightarrow{\varepsilon}^* w_1 \xrightarrow{a} u_1$ and $(u_1, u_2) \in \mathcal{X}$; (2) if $(v_1, v_2) \in \mathcal{X}$, for any path $v_1 \xrightarrow{\varepsilon}^* u_1$ such that $(u_1, \&x) \in O$, there is a path $v_2 \xrightarrow{\varepsilon}^* u_2$ such that $(u_2, \&x) \in O$, and conversely, for any path $v_2 \xrightarrow{\varepsilon}^* u_2$ such that $(u_2, \&x) \in O$, there is a path $v_1 \xrightarrow{\varepsilon}^* u_1$ such that $(u_1, \&x) \in O$; and (3) $\text{dom}(I_1) = \text{dom}(I_2)$ and $(I_1(\&x), I_2(\&x)) \in \mathcal{X}$ for any $\&x \in \text{dom}(I_1) = \text{dom}(I_2)$. 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 .

$$\begin{aligned} g ::= & \{ \} \mid \{ a : g \} \mid g_1 \cup g_2 && \text{(graph constructors)} \\ & \mid \&x \triangleright g \mid \&y \mid () \mid g_1 \oplus g_2 \\ & \mid g_1 @ g_2 \mid \text{cycle}(g) && \text{(structural recursion)} \\ a ::= & l \mid \mathbf{a} && \text{(graph variable)} \end{aligned}$$

Figure 3. The syntax of the positive subset of UnCAL

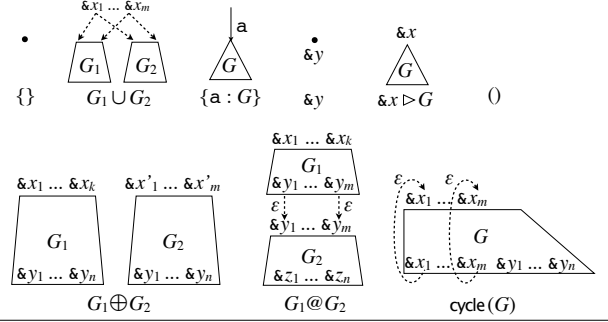


Figure 4. Graph Constructors

Note that the graph bisimulation is different from weak bisimulation [37] or the language equivalence of automata, as demonstrated by the following examples.



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.

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, a careful discussion is needed for *isEmpty*, as there is no computable counterpart of *isEmpty* in *general* functional programs, and it must be converted to an productivity-test oracle (Section 4). However, unlike that in Section 4, the discussions in Sections 5, 6 and 7 can be easily extended to *isEmpty* because *isEmpty* becomes computable for FUnCAL programs thanks to the type system in Section 6.

3.2.1 Graph Constructors

UnCAL has the nine graph constructors, $\{ \}$, $\{ _ : _ \}$, \cup , $\&x \triangleright _$, $\&y$, $()$, \oplus , $@$, and *cycle*. Some of them are already mentioned in Section 1.1. A record notation shown in Section 1.1 such as $\{\text{name} : \text{Alice}, \text{email} : \text{alice}\}$ is a syntax sugar for $\{\text{name} : \{\text{Alice} : \{ \} \} \} \cup \{\text{email} : \{\text{alice} : \{ \} \} \}$. 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 $\llbracket \{ \} \rrbracket$ is defined by:

$$\llbracket \{ \} \rrbracket = (\{v\}, \emptyset, \{ \& \mapsto v \}, \emptyset)$$

Here v is a fresh node.

Edge Extension The expression $\{a : g\}$ constructs a graph by adding an edge with label a pointing to the root of the graph $\llbracket g \rrbracket$; formally, its semantics is defined by:

$$\llbracket \{a : g\} \rrbracket = (V \cup \{u\}, E \cup \{(u, a, v)\}, \{\& \mapsto u\}, O) \\ \text{where } (V, E, \{\& \mapsto v\}, O) = \llbracket g \rrbracket$$

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

$$\llbracket g_1 \cup g_2 \rrbracket = (V_1 \cup V_2 \cup \{v\}, E, \{\& \mapsto v\}, O_1 \cup O_2) \\ \text{where } (V_1, E_1, I_1, O_1) = \llbracket g_1 \rrbracket \\ (V_2, E_2, I_2, O_2) = \llbracket g_2 \rrbracket \\ E = E_1 \cup E_2 \cup \{(v, \varepsilon, I_1(\&)), (v, \varepsilon, I_2(\&))\}$$

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:

$$\llbracket \&y \rrbracket = (\{v\}, \emptyset, \{\& \mapsto v\}, \{(v, \&y)\})$$

Here v is a fresh node.

Naming Root The expression $\&x \triangleright g$ names the root of $\llbracket g \rrbracket$ by $\&x$, of which semantics is defined by:

$$\llbracket \&x \triangleright g \rrbracket = (V, E, \{\&x \mapsto v\}, O) \\ \text{where } (V, E, \{\& \mapsto v\}, O) = \llbracket g \rrbracket$$

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 $\llbracket g_1 \rrbracket$ and $\llbracket g_2 \rrbracket$ with different sets of input markers, of which semantics is defined by:

$$\llbracket g_1 \oplus g_2 \rrbracket = (V_1 \cup V_2, E_1 \cup E_2, I_1 \cup I_2, O_1 \cup O_2) \\ \text{where } (V_1, E_1, I_1, O_1) = \llbracket g_1 \rrbracket \\ (V_2, E_2, I_2, O_2) = \llbracket g_2 \rrbracket$$

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

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

$$\llbracket () \rrbracket = (\emptyset, \emptyset, \emptyset, \emptyset)$$

Plugging In The expression $g_1 @ g_2$ replaces holes in $\llbracket g_1 \rrbracket$ with roots of $\llbracket g_2 \rrbracket$ that share the same names, of which semantics is defined by:

$$\llbracket g_1 @ g_2 \rrbracket = (V_1 \cup V_2, E, I_1, O_2) \\ \text{where } (V_1, E_1, I_1, O_1) = \llbracket g_1 \rrbracket \\ (V_2, E_2, I_2, O_2) = \llbracket g_2 \rrbracket \\ E = E_1 \cup E_2 \cup \{(v, \varepsilon, I_2(\&x)) \mid (v, \&x) \in O_1\}$$

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

⁴In 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 $\&x_1, \dots, \&x_n$, the original $g_1 \cup g_2$ can be rewritten as $(\&x_1 \triangleright ((\&x_1 @ g_1) \cup (\&x_1 @ g_2))) \oplus \dots \oplus (\&x_n \triangleright ((\&x_n @ g_1) \cup (\&x_n @ g_2)))$, in which \cup satisfies this restriction.

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

$$\llbracket \text{cycle } g \rrbracket = (V, E', I, \{(v, \&x) \in O \mid \&x \notin \text{dom}(I)\}) \\ \text{where } (V, E, I, O) = \llbracket g \rrbracket \\ E' = E \cup \{(v, \varepsilon, I(\&x)) \mid (v, \&x) \in O\}$$

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

$$\&a @ \text{cycle}((\&a \triangleright (\{\text{name} : \{\text{Alice} : \{\}\}\} \cup \{\text{friend} : \&b\}\})) \\ \oplus (\&b \triangleright (\{\text{name} : \{\text{Bob} : \{\}\}\} \cup \{\text{friend} : \&a\}\})))$$

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

$$f(\{\}) = \{\} \quad (\text{SR1})$$

$$f(\{a : G\}) = g[a/l, G/t] @ f(G) \quad (\text{SR2})$$

$$f(G_1 \cup G_2) = f(G_1) \cup f(G_2) \quad (\text{SR3})$$

Thanks to **srec**, UnCAL can express many graph transformations in an efficient way, with guarantee of termination [8, 26].

Formally, its semantics (the bulk semantics [8]) is defined by:

$$\llbracket \text{srec}(\lambda(l, t).g)(g') \rrbracket = (V' \cup \bigcup_{\zeta \in E} V_\zeta, E' \cup \bigcup_{\zeta \in E} E_\zeta, \\ \{\&x \mapsto u_{v, \&x} \mid \&x \in Z\}, \emptyset)$$

where

$$(V, E, \{\& \mapsto v_0\}, \emptyset) = \llbracket g' \rrbracket$$

$$(V_\zeta, E_\zeta, I_\zeta, O_\zeta) = \llbracket g[a/l, (V, E, \{\& \mapsto v\}, \emptyset)/t] \rrbracket \\ (\zeta = (_, a, v))$$

$$V' = \{u_{v, \&x} \mid v \in V, \&x \in Z\}$$

$$E' = \{(v', \varepsilon, u_{v, \&x}) \mid \exists a, u. (v', \&x) \in O_{(u, a, v)}\} \\ \cup \{(u_{v, \&x}, \varepsilon, v') \mid \exists a, u. I_{(v, a, u)}(\&x) = v'\}$$

Here, V' are fresh nodes, and $Z = \text{dom}(I_\zeta)$ and $\text{ran}(O_\zeta) \subseteq Z$ for each $\zeta \in E$. We assume that $\text{dom}(I_\zeta) = \text{dom}(I_{\zeta'})$ for all $\zeta, \zeta' \in E$, and V_ζ and $V_{\zeta'}$ are disjoint for different $\zeta, \zeta' \in E$. Intuitively, the semantics computes $g[a/l, G/t]$ for each edge in $\llbracket g' \rrbracket$, and connects them by ε -edges which corresponds to (SR2) and (SR3). Unlike the original definition [8], we require the graph $\llbracket g' \rrbracket$ 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 have noticed 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, 24], in which a graph type is of the form DB_Y^X . A type DB_Y^X 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_Y^X 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

This section formally describes our translation from UnCAL programs to functional ones, whose idea has been roughly explained in Section 1.1. The translation enables us to reason about UnCAL

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

Figure 5. The syntax of FUnCAL

programs as functional ones, which would be useful to import verification techniques for functional programs (Section 2).

The idea of the translation is to convert an UnCAL program concerning graphs to a functional one concerning 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.

Here, we allow that a translated functional program may not terminate; for example, an UnCAL expression $\text{cycle}(\&)$ satisfying

$$\llbracket \text{cycle}(\&) \rrbracket = \text{graph} \sim \llbracket \{\} \rrbracket$$

is converted to $\text{fix}_{\mathbf{G}}(\lambda x.x)$ 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 $\text{srec}(\dots)(g)$ 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, $\text{srec}(\lambda(l, t).g')(\&y) = \&y$ holds for g' with type $\text{DB}_{\{\&\}}^{\{\&\}}$. 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 **srec**.

4.1 Translation

The syntax of the target language of our translation is given in Figure 5. We call the language FUnCAL; at this point, we assume the standard call-by-name semantics for it. FUnCAL contains λ -expressions, tuples (where π_i^n is the projection of the i th element from an n tuple), (infinite) tree constructors, and the (first-order) fixed-point operator $\text{fix}_{\mathbf{G}}$, and structural recursions $\text{fold}_n f$. 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 $\text{fix}_{\mathbf{G}}$ instead of **letrec** that appeared in Section 1.1, since it is handy to discuss reductions. The structural recursion $\text{fold}_n f$ is “fold” for the tree constructors defined recursively as:

$$\begin{aligned} \text{fold}_n f \bullet &= (\bullet, \dots, \bullet) & (\text{Fold1}) \\ \text{fold}_n f (x : y) &= f x (\text{fold}_n f y) & (\text{Fold2}) \\ \text{fold}_n f (x \cup y) &= (\text{fold}_n f x) \cup (\text{fold}_n f y) & (\text{Fold3}) \end{aligned}$$

Note that $\text{fold}_n f$ returns an n -tuple of trees rather than a tree; in the right-hand side of (Fold3), we overload \cup to tuples as $(x_1, \dots, x_n) \cup (y_1, \dots, y_n) = (x_1 \cup y_1, \dots, x_n \cup y_n)$. 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 $g : \text{DB}_Y^X$ is translated to

$$e : \mathbf{G}^{|Y|} \rightarrow \mathbf{G}^{|X|}$$

where \mathbf{G} is the (coinductive) datatype defined by:

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

At this point, we assume that FUnCAL has the standard simple type system with the datatype \mathbf{G} and the label type \mathbf{L} ; we later refine it

to guarantee termination (Section 6). For example, an expression $\lambda f. \text{fold}_n f$ has type $(\mathbf{L} \rightarrow \mathbf{G}^n \rightarrow \mathbf{G}^n) \rightarrow \mathbf{G} \rightarrow \mathbf{G}^n$.

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

Our translation is defined according to the typing derivation of UnCAL. A translation judgment $\Gamma \vdash g : \text{DB}_Y^X \rightsquigarrow e$ reads that an expression g of type DB_Y^X under a typing environment (i.e., a mapping from graph/label variables to types) Γ in UnCAL is converted to an expression e , where the type of g and types in Γ are converted from DB_Y^X to $\mathbf{G}^{|Y|} \rightarrow \mathbf{G}^{|X|}$. Figure 6 shows the translation rules for the UnCAL graph constructors. If we ignore the $\rightsquigarrow e$ part of $\Gamma \vdash g : \text{DB}_Y^X \rightsquigarrow e$, the judgment and rules coincide to the typing judgment and rules of UnCAL [6, 24].

The conversion rule for **srec** is a bit involved and thus is written separately as follows.

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

Here, $\text{para}_n e$, representing a “paramorphism”⁵ [35], where an expression $\lambda f.\text{para}_n f$ has type $(\mathbf{L} \rightarrow \mathbf{G} \rightarrow \mathbf{G}^n \rightarrow \mathbf{G}^n) \rightarrow \mathbf{G} \rightarrow \mathbf{G}^n$, is a syntax sugar defined by:

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

where $p : \mathbf{G}^{n+1} \rightarrow \mathbf{G}^n$, $p' : \mathbf{G}^n \rightarrow \mathbf{G}$ and $q : \mathbf{G}^n \rightarrow \mathbf{G} \rightarrow \mathbf{G}^{n+1}$ are functions to rearrange tuples defined as $p x = (\pi_1 x, \dots, \pi_n x)$, $p' x = \pi_{n+1} x$, and $q x y = (\pi_1 x, \dots, \pi_n x, y)$. This definition of para_n is similar to how a paramorphism is represented by a catamorphism (fold) via tupling [35]. The rule C-REC becomes a bit complicated due to explicit conversions between \mathbf{G} and $() \rightarrow \mathbf{G}$. Since the argument of para_n must be of type \mathbf{G} instead of $() \rightarrow \mathbf{G}$, we apply $()$ to e_2 . In addition, since the conversion assumes that t in e_1 has type $() \rightarrow \mathbf{G}$, we construct such a function by $\lambda().t'$.

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

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_Y^X , which can contain output markers in Y , is translated to a tree-to-tree function $\mathbf{G}^{|Y|} \rightarrow \mathbf{G}^{|X|}$ 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 [37].

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

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

$$\begin{array}{c}
\frac{}{\Gamma \vdash a : L \rightsquigarrow a} \text{C-LAB} \quad \frac{}{\Gamma \vdash \{\} : \text{DB}_Y^{\{\&\}} \rightsquigarrow \bullet} \text{C-SINGLE} \\
\frac{\Gamma \vdash g : \text{DB}_Y^{\{\&\}} \rightsquigarrow e \quad \Gamma \vdash a : L \rightsquigarrow a'}{\Gamma \vdash \{a : g\} : \text{DB}_Y^{\{\&\}} \rightsquigarrow \lambda y. a' : (e \ y)} \text{C-EDGE} \\
\frac{\{\Gamma \vdash g_i : \text{DB}_Y^{\{\&\}} \rightsquigarrow e_i\}_{i=1,2}}{\Gamma \vdash g_1 \cup g_2 : \text{DB}_Y^{\{\&\}} \rightsquigarrow \lambda y. (e_1 \ y) \cup (e_2 \ y)} \text{C-UNI} \\
\frac{\Gamma \vdash g : \text{DB}_Y^{\{\&\}} \rightsquigarrow e}{\Gamma \vdash \&x \triangleright g : \text{DB}_Y^{\{\&x\}} \rightsquigarrow e} \text{C-ROOT} \quad \frac{}{\Gamma \vdash () : \text{DB}_Y^{\emptyset} \rightsquigarrow \lambda y. ()} \text{C-EMP} \\
\frac{\overline{Y} = (\&y_1, \dots, \&y_n) \quad \&y = \&y_i \quad 1 \leq i \leq n}{\Gamma \vdash \&y : \text{DB}_Y^{\{\&\}} \rightsquigarrow \lambda z. \pi_i z} \text{C-HOLE} \\
\frac{\{\Gamma \vdash g_i : \text{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 \overline{X}_2}{\Gamma \vdash g_1 \oplus g_2 : \text{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 : \text{DB}_Y^X \rightsquigarrow e_1 \quad \Gamma \vdash g_2 : \text{DB}_Z^Y \rightsquigarrow e_2}{\Gamma \vdash g_1 @ g_2 : \text{DB}_Z^X \rightsquigarrow \lambda y. e_1 \ (e_2 \ y)} \text{C-SUBST} \\
\frac{\Gamma \vdash g : \text{DB}_{X \uplus Y}^X \rightsquigarrow e}{\Gamma \vdash \text{cycle}(g) : \text{DB}_Y^X \rightsquigarrow \lambda \overline{Y}. \text{fix}_G(\overline{X}. e \ \overline{X} \cup \overline{Y})} \text{C-CYC} \\
\frac{z : \text{graph/label variable}}{\Gamma \vdash z : \Gamma(z) \rightsquigarrow z} \text{C-VAR} \quad \frac{\Gamma \vdash g : \text{DB}_Y^X \rightsquigarrow e \quad Y \subseteq Y'}{\Gamma \vdash g : \text{DB}_{Y'}^X \rightsquigarrow \lambda \overline{Y}. e \ \overline{Y}} \text{C-SUB}
\end{array}$$

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, $E = \{(e, a, e') \mid e \Rightarrow^* a : e'\}$, and $I = \{\&x_1 \mapsto \pi_1 e_0, \dots, \&x_n \mapsto \pi_n e_0\}$. Here, the relation (\Rightarrow) is defined by:

$$e_1 \cup e_2 \Rightarrow e_1 \quad e_1 \cup e_2 \Rightarrow e_2 \quad e \Rightarrow e' \text{ if } e \rightarrow e'$$

where, \rightarrow is the call-by-name reduction. \square

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. 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 $G :: \text{DB}_Y^X$ and an expression $e :: \mathbb{G}^{|Y|} \rightarrow \mathbb{G}^{|X|}$ by: $G \approx e$ iff $(G @ (G_1 \oplus \dots \oplus G_{|Y|})) \sim (e \ (e_1, \dots, e_{|Y|}))$ for any G_j, e_j ($1 \leq j \leq |Y|$) such that $G_j :: \text{DB}_{\emptyset}^{\emptyset}$ and $G_j \sim e_j$.

Now, we have the following theorem.

Theorem 1 (Correctness). *If $\vdash g : \text{DB}_Y^X \rightsquigarrow e$, $\llbracket g \rrbracket \approx e$ holds.*

Proof. See Appendix A. \square

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 [26].

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 $e \Rightarrow^* a : e'$ for some 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 at this point. 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.

In contrast, 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. Graph Transformation as Graph Reduction

This section gives a semantics of FUnCAL so that we can obtain *finite graphs* rather than infinite trees after evaluation. Our semantics is basically a “zipper”ed abstract machine of Nakata and Hasegawa [38]’s lazy semantics, which extends Launchbury [32]’s natural semantics with the *black hole* [2, 3, 38] (“apparent undefinedness”). This section focuses on the formal description of the semantics. The discussions on termination are 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 $\text{fix}_G(\lambda x. a : x)$ in the usual lazy evaluation is cyclic and has a similar structure to a corresponding UnCAL graph $\llbracket \text{cycle}(\{a : \&\}) \rrbracket$. However, an extra effort is required to handle $\text{fix}_G(\lambda x. x)$ and $\text{fold}(\lambda a. \lambda r. r)(e)$ for example, which are nonterminating in usual semantics.

The lazy semantics with the black hole [38] plays an important role to resolve this problem. Since $\text{fix}_G(\lambda x. x)$ evaluates to the black hole without running infinitely in this lazy semantics, we identify the black hole with a singleton graph, and obtain a singleton graph as the evaluation result of $\text{fix}_G(\lambda x. x)$. Still, the semantics is not sufficient for terminating evaluation of recursions such as $\text{fold}(\lambda a. \lambda r. r)(e)$. To make the evaluation of recursions terminating, we adopt *memoization*. Roughly speaking, since e of $\text{fold}(\lambda a. \lambda r. r)(e)$ represents a graph, the recursive call of $\text{fold}(\lambda a. \lambda r. r)$ must visit 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

To adopt memoization for structural recursions, we change the FUnCAL syntax as

$$e ::= \dots \mid \text{fold}^M e$$

where $\text{fold}_n e$ is replaced with $\text{fold}^M e$ 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 $\text{fold}^M e$ above does not have the subscript.

In what follows, we shall use a metavariable C for binary constructors “ \cdot ” and “ \cup ”.

5.2 Abstract Machine

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

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

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

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

$$E ::= \square \mid E \ e \mid (\text{fold}^M e) \ E \mid x := E$$

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

Figure 7. Reduction rules of our lazy abstract machine: we assume that $(\lambda x. e)$ is α -renamed to conform with the freshness condition.

An evaluation context $x := \square$ is a key of 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 \square with e . 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 $\langle x \mid x_1 = e_1, \dots, x_n = e_n \rangle$ as **letrec** $x_1 = e_1$ **and** \dots **and** $x_n = e_n$ **in** x . We sometimes write $\langle e \mid \mu \rangle$ for $\langle x \mid x = e, \mu \rangle$ where we do not care x .

The reduction relation $\langle x \mid \mu \rangle \rightarrow \langle x \mid \mu' \rangle$ is defined by the rules in Figure 7. The rules except the ones for $\text{fold}^M e$ are just straightforward extensions of [38] with constructors.

The black hole \bullet represents the apparent undefinedness yielded when the value of x is required by the evaluation of x itself [38]. A typical example is $\text{fix}_G (\lambda x. x)$, which will be evaluated as:

$$\begin{aligned}
\langle \text{fix}_G (\lambda x. x) \mid \emptyset \rangle &\rightarrow \langle w \mid w = (\lambda x. x) w \rangle \\
&\rightarrow \langle w := (\lambda x. x) w \mid w = \bullet \rangle \\
&\rightarrow \langle w := x \mid w = \bullet, x = w \rangle \\
&\rightarrow \langle w := (x := w) \mid w = \bullet, x = \bullet \rangle \\
&\rightarrow^* \langle w := (x := \bullet) \mid w = \bullet, x = \bullet \rangle \\
&\rightarrow^* \langle \bullet \mid w = \bullet, x = \bullet \rangle
\end{aligned}$$

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

$$\begin{aligned}
\langle \text{fix}_G (\lambda x. a : x) \mid \emptyset \rangle &\rightarrow \langle w \mid w = (\lambda x. a : x) w \rangle \\
&\rightarrow \langle w := ((\lambda x. a : x) w) \mid w = \bullet \rangle \\
&\rightarrow^* \langle w := a' : x' \mid a' = a, x' = x, x = w, w = \bullet \rangle \\
&\rightarrow \langle a' : x' \mid a' = a, x' = x, x = w, w = a' : x' \rangle
\end{aligned}$$

The reduction rules for $\text{fold}^M e$ are keys in this semantics. It basically works as $\text{fold}_1 e$ 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 $\text{fold}^M e$ looks up the memo if there is corresponding entry in the memo. By memoization, the problematic example in Section 1 that eliminates all the edges evaluates to \bullet without major changes to the original semantics [38], as illustrated below.

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

$$\text{fold}^0 (\lambda z. \lambda r. r) (\text{fix}_G (\lambda x. a : x))$$

Here, $\text{fold}^0 (\lambda z. \lambda r. r)$ eliminates all the edges, and thus we expect that the expression results in \bullet (a singleton graph). Let us write el^M for $\text{fold}^M (\lambda z. \lambda r. r)$, then the above expression is evaluated as:

$$\begin{aligned}
&\langle el^0 (\text{fix}_G (\lambda x. a : x)) \mid \emptyset \rangle \\
&\rightarrow^* \langle el^0 (a' : x') \mid a' = a, x' = x, x = w, w = a' : x', \dots \rangle \\
&\rightarrow \langle u \mid u = (\lambda z. \lambda r. r) a' (el^{\{(a' : x') \mapsto u\}} x'), \dots \rangle \\
&\rightarrow^* \langle u := el^{\{(a' : x') \mapsto u\}} x' \mid u = \bullet, \dots \rangle \\
&\rightarrow^* \langle u := el^{\{(a' : x') \mapsto u\}} (a' : x') \mid u = \bullet, \dots \rangle \\
&\rightarrow \langle u := u \mid u = \bullet, \dots \rangle \quad \{\text{hit}\} \\
&\rightarrow \langle \bullet \mid \dots \rangle.
\end{aligned}$$

Here, we underlined the configurations where el^M inspected the memo M . Memoization plays an important role to obtain this 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. \square

An important property on memos is that, the looked-up objects are always values, which will be used in Section 7.

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

5.3 Extracting Graphs

After an evaluation, we extract a graph from the “graph” structure of a heap. For example, for a configuration $\langle x \mid x = a : x, a = a \rangle$, we obtain a graph $G = (V, E, I, O)$ with $V = \{x\}$, $E = \{(x, a, x)\}$, $I = \{\& \mapsto x\}$ and $O = \emptyset$. In general, a heap may contain unevaluated expressions. If a heap contains unevaluated expressions as $\{x = (\lambda y. y)(a : x)\}$, 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 a variable x *accessible* from y in μ if (x, y) belongs to the reflexive transitive closure of the $\{(z, w) \mid w \in \text{fv}(\mu(z))\}$. In this subsection and in Section 7, we shall even omit (\cup) for simplicity.

This deep evaluation is done easily: for a configuration $\langle x \mid \mu \rangle$, we just evaluate $\langle \text{elim}^0 x \mid \mu \rangle$ where $\text{elim}^M = \text{fold}^M (\lambda a. \lambda r. \text{if } a = a \text{ then } r \text{ else } r)$. Here, **if** is used just to evaluate a ; extending the abstract machine to **if** is straightforward. The evaluation of application $\text{elim}^0 x$ eliminates all the edges from x by evaluating all accessible variables from x , and results in \bullet if terminates.

Now, we are ready to define the graph extraction formally. Let e_0 be a closed expression of type G , and suppose that we have $\langle \text{elim}^0 x_0 \mid x_0 = e_0 \rangle \rightarrow^* \langle \bullet \mid \mu \rangle$. Then, $\mu(x)$ is a value for every x accessible from x_0 . After that, the graph extraction is done easily. This process is summarized as $\text{graphify}(x_0, e_0)$ below.

$$\begin{aligned}
\text{graphify}(x_0, e_0) &= (V, E, \{\& \mapsto x_0\}, \emptyset) \\
&\text{where } V = \text{accessible variables from } x \text{ in } \mu \\
&\quad E = \bigcup_{x=(x_1 : x_2) \in \mu, x \in V} \{(x, \mu(x_1), x_2)\} \\
&\quad \langle \text{elim}^0 x_0 \mid x_0 = e_0 \rangle \rightarrow^* \langle \bullet \mid \mu \rangle
\end{aligned}$$

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

Thus, the termination under the context $\text{elim}^0 \square$ means that an expression corresponds to a finite graph.

Treatment of (\cup) . If we have (\cup) , it suffices to use a context $\text{isEmpty}^0 (\text{elim}^0 \square)$ instead of $\text{elim}^0 \square$, where isEmpty^M is a memoized version of isEmpty . If we only allow isEmpty^0 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 $\text{graphify}(x_0, e_0)$ is changed accordingly; specifically, the definition of E is changed to $E = \dots \cup \bigcup_{x=(x_1 \cup x_2) \in \mu, x \in V} \{(x, \varepsilon, x_1), (x, \varepsilon, x_2)\}$.

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

Theorem 2. *If $\text{graphify}(x, e) = G$, 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 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 affect termination, which will be used in Section 7.

Lemma 2 (Memo and Termination). *If $\langle \text{elim}^0 e \mid \mu \rangle$ terminates, then $\langle \text{elim}^M e \mid \mu \rangle$ also terminates for any M such that $\mu(M(v))$ is a value for all $v \in \text{dom}(M)$.* \square

6. Type System

In this section describes the type system that guarantees termination of \rightarrow under the context elim^0 . That is, in this type system, well-typed expressions represent *finite*-graph transformations.

6.1 Idea

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

$$\begin{aligned} bs &= \text{fix}_G (\lambda x. b : x) \\ aInB &= \text{fold} (\lambda z. \lambda r. z : \text{insA } r) \\ \text{insA} &= \text{fold} (\lambda z. \lambda r. a : z : r). \end{aligned}$$

(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

$$\begin{aligned} aInB \ bs &\rightarrow^* b : \text{insA } (aInB \ bs) \\ &\rightarrow^* b : \text{insA } (b : \text{insA } (aInB \ bs)) \\ &\rightarrow^* b : a : b : \text{insA}^2 (aInB \ bs) \\ &\rightarrow^* b : a : b : a : a : b : \text{insA}^3 (aInB \ bs) \\ &\rightarrow^* b : a : b : a : a : a : b : \dots : \text{insA}^n (aInB \ bs) \end{aligned}$$

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 $\text{elim}^0 (aInB \ bs)$. 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

$$el \ bs \rightarrow^* el \ bs \quad \text{insA } bs \rightarrow^* a : b : \text{insA } bs$$

Thanks to this looping structure, $\text{elim}^0 (el \ bs)$ and $\text{elim}^0 (\text{insA } bs)$ terminate with memoization.

To exclude such a problematic case, we overapproximate the number of nested applications of folds and bound it by typing. Specifically, we associate a graph type with an integer as $G\langle n \rangle$, and the number is increased by fold, as $\text{fold}^M e : G\langle n \rangle \rightarrow G\langle m \rangle$

$$\begin{array}{c} \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \text{T-VAR} \quad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x. e : \tau_1 \rightarrow \tau_2} \text{T-ABS} \\ \frac{\Gamma \vdash e_1 : \tau' \rightarrow \tau \quad \Gamma \vdash e_2 :: \tau'}{\Gamma \vdash e_1 e_2 : \tau} \text{T-APP} \\ \frac{}{\Gamma \vdash a : L} \text{T-LABEL} \quad \frac{\Gamma \vdash e_1 : L \quad \Gamma \vdash e_2 : G\langle n \rangle}{\Gamma \vdash (e_1 : e_2) : G\langle n \rangle} \text{T-CONS} \\ \frac{\{\Gamma \vdash e_i : G\langle n \rangle\}_{i=1,2}}{\Gamma \vdash e_1 \cup e_2 : G\langle n \rangle} \text{T-CHOICE} \quad \frac{}{\Gamma \vdash \bullet : \tau} \text{T-BH} \\ \frac{\Gamma \vdash e : G\langle n \rangle \rightarrow G\langle n \rangle}{\Gamma \vdash \text{fix}_G e : G\langle n \rangle} \text{T-FIX} \\ \frac{\Gamma \vdash e : L \rightarrow G\langle m \rangle \rightarrow G\langle m \rangle \quad \{\Gamma \vdash v : G\langle n \rangle \wedge \Gamma \vdash x : G\langle m \rangle\}_{(v,x) \in M} \quad m > n}{\Gamma \vdash \text{fold}^M e : G\langle n \rangle \rightarrow G\langle m \rangle} \text{T-CATA} \\ \frac{\Gamma \vdash e : \tau' \quad \tau' \preceq \tau}{\Gamma \vdash e : \tau} \text{T-SUB} \end{array}$$

Figure 8. Typing rules for termination.

where $e : L \rightarrow G\langle m \rangle \rightarrow G\langle m \rangle$ and $m > n$. We call the number *generation*. Note that generations are integers instead of natural numbers because they are more convenient for proving Theorem 3. For example, $aInB = \text{fold}^0 (\lambda z. \lambda r. z : \text{insA } r)$ is ill-typed; the body of fold must have the type $L \rightarrow G\langle m \rangle \rightarrow G\langle m \rangle$ but it can only have the type $L \rightarrow G\langle m \rangle \rightarrow G\langle m + l \rangle$ with $l > 0$ because insA increase the generation at least by one. In contrast, insA itself and el are well-typed. We will add similar restrictions on fix_G because fix_G can also produce non-regular results,

6.2 Types

A type τ is defined as follows.

$$\tau ::= G\langle n \rangle \mid \tau_1 \rightarrow \tau_2 \mid L \quad (n \in \mathbb{Z})$$

Types consist of graph types with generation ($G\langle n \rangle$), function types (\rightarrow) and the label type (L). It is natural to have a subtyping relation $G\langle m \rangle \preceq G\langle n \rangle$ with $m \leq n$ because generations are overapproximation of number of nested applications of fold. 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 λ s are standard ones. The rules T-CONS, T-CHOICE and T-FIX says that the tree constructors construct a graph of a generation from graphs of the same generations. The rule T-BH says that \bullet is something similar to an exception. The rule T-CATA is special in our type system, which says that the resulting graph of $\text{fold}^M f$ must be a strictly newer generation than its argument. The resulting graph not necessary be 1-generation newer because f can use other folds many times. Also, note that we typecheck memos in the rule to ensure the preservation property. Our type system can be easily extended to configurations, of which preservation property can also be proved easily.

An important property is that the translation in Section 4 yields well-typed programs from well-typed UnCAL.

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

Proof. See Appendix B. \square

Notice that this translation uses general versions of $\text{fix}_G e$ and $\text{fold}_k^M e$ extended for tuples. Here, we just note fold_k^M 's general typing rule; that for $\text{fix}_G e$ is much straightforward.

$$\frac{\Gamma \vdash e : L \rightarrow \prod_{1 \leq i \leq k} G\langle m_i \rangle \rightarrow \prod_{1 \leq i \leq n} G\langle m_i \rangle \quad \{\Gamma \vdash v : G\langle n \rangle \wedge \Gamma \vdash x : \prod_{1 \leq i \leq k} G\langle m_i \rangle\}_{(v,x) \in M} \quad k < \min \{m_i\}_{1 \leq i \leq k}}{\Gamma \vdash \text{fold}_k^M e : G\langle n \rangle \rightarrow \prod_{1 \leq i \leq k} G\langle m_i \rangle} \text{T-CATA}$$

Here, we write $\prod_{1 \leq i \leq k} \tau_i$ for $\tau_1 \times \dots \times \tau_k$. This type allows us to traverse the i th component of a tuple to produce the j th component if $m_i < m_j$, as $\text{fold}_2^0(\lambda a. \lambda r. (\text{fold}_2^0 f (\pi_2 z), a : \pi_2 z))$, which is a key to implement para_{k-1} expressive enough to represent UnCAL's **srec**. For example, the translation of the query in Introduction needs such a traversal. This is the key difference from Nishimura and Ohori [39]'s type system that distinguishes types for graphs (traversable) from types for nodes (untraversable).

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 $\vdash f : G\langle n \rangle \rightarrow G\langle m \rangle$. If $\vdash e_i : G\langle n \rangle$ ($i = 1, 2$), $e_1 \sim e_2$ implies $f e_1 \sim f e_2$.*

Proof. See Appendix C. \square

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 $\text{elim}^\theta e$ terminates and thus graphify succeeds. To simplify our discussion, as wrote in Section 5.3, we shall ignore (\cup) in this section.

Formally, we will prove the following type soundness theorem by using logical relation [44].

Theorem 5. *If $\vdash e : G\langle n \rangle$, then $\langle \text{elim}^\theta x \mid x = e \rangle \rightarrow^* \langle \bullet \mid \mu \rangle$.*

7.1 Typing Configurations without Cyclic Dependency

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 [44]. 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 $\text{dom}(\mu)$ and $\text{dom}(\Gamma)$ are disjoint, and thus we can deterministically 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 $\text{dom}(\Delta)$, then $\Gamma \vdash_\mu e : \tau$ holds.* \square

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 $G\langle n \rangle$ cannot “observe” graphs of type $G\langle n \rangle$; recall that fold is the only language construct that can observe graphs.

Lemma 4. *Let $Z = \{z_1, \dots, z_k\}$. Suppose $z_1 : G\langle n \rangle, \dots, z_k : G\langle n \rangle \vdash_\mu e : G\langle n \rangle$. Then, if $\langle \text{elim}^M e\sigma \mid \mu\sigma \rangle$ terminates for some $\sigma \in Z \rightarrow V_M$, then $\langle \text{elim}^M e\sigma' \mid \mu\sigma' \rangle$ terminates for all $\sigma' \in Z \rightarrow V_M$ where $V_M = \text{dom}(M) \cup \{\bullet\}$.*

Proof. See Appendix D. \square

7.2 Logical Relation

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

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

- $\langle e \mid \mu \rangle \in \mathcal{R}_L$ iff $\langle e \mid \mu \rangle \rightarrow^* \langle v \mid \mu' \rangle$ for some v and μ' .
- $\langle e \mid \mu \rangle \in \mathcal{R}_{G\langle n \rangle}$ iff $\langle \text{elim}^\theta e \mid \mu \rangle \rightarrow^* \langle \bullet \mid \mu' \rangle$ for some μ' .
- $\langle f \mid \mu \rangle \in \mathcal{R}_{\tau_1 \rightarrow \tau_2}$ iff $\langle f \mid \mu \rangle \rightarrow^* \langle v \mid \mu' \rangle$ 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 $x \in \text{dom}(\mu) \cap \text{dom}(\mu')$. Intuitively, $\mathcal{R}(\tau)$ defines pairs of expressions and heaps that are “meaningful” as *finite-graph* transformations. Especially, $\langle e \mid \mu \rangle \in \mathcal{R}_{G\langle n \rangle}$ means that $\langle e \mid \mu \rangle$ corresponds to a finite graph. Note that, thanks to elim^θ in the definition, we have $a\text{In}B \text{ bs} \notin \mathcal{R}_{G\langle n \rangle}$ for $\text{bs} = \text{fix}_G(\lambda x. b : x)$ while the evaluation of $a\text{In}B \text{ bs}$ itself terminates. Also note that we have $\mathcal{R}_{G\langle n \rangle} = \mathcal{R}_{G\langle m \rangle}$ because $\vdash_\mu e : G\langle n \rangle$ if and only if $\vdash_\mu e : G\langle m \rangle$; recall that the generations are integers, and only their differences matter (cf. Lemma 4).

In the later proof, we will use the following properties on \mathcal{R} . Lemma 5 says that “garbage cells” in the heap do 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 $\langle e \mid \mu \cup \mu' \rangle \in \mathcal{R}_\tau$.* \square

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

7.3 Lemmas for Recursive Definitions

In advance of 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 $\text{fix}_G e$ expressions corresponds to a finite graph if e preserves finiteness.

Lemma 7 (fix_G). *If $\vdash_\mu e_0 :: G\langle n \rangle \rightarrow G\langle n \rangle$ and $\langle e_0 e' \mid \mu \cup \mu' \rangle \in \mathcal{R}_{G\langle n \rangle}$ for any $\langle e' \mid \mu' \rangle \in \mathcal{R}_{G\langle n \rangle}$, then $\langle \text{fix}_G e_0 \mid \mu \rangle \in \mathcal{R}_{G\langle n \rangle}$ holds.*

Proof. It suffices to show that $\langle \text{elim}^\theta w \mid w = e_0 w, \mu \rangle$ terminates. Consider a configuration $c_0 = \langle \text{elim}^\theta w \mid w = e_0 u, u = \bullet, \mu \rangle$ where fix_G is unfolded only once. We prove the statement by showing that $d_0 = \langle \text{elim}^\theta w \mid w = e_0 w, \mu \rangle$ terminates if c_0 does. Then, since we have the termination of c_0 by the premise of this lemma, 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:

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

Accordingly, we must have the following sequence.

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

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

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

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

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

Since $\bullet, v \in \text{dom}(M)$, from Lemma 4, the reductions from $\langle \text{elim}^M (E[\bullet]) \mid \eta_* \rangle$ and $\langle \text{elim}^M (E[w/u][v]) \mid \eta \rangle$ 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 $\text{fold}^M e$ 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). *If $\vdash_\mu \text{fold}^M e :: G\langle n \rangle \rightarrow G\langle m \rangle$ ($m > n$) and $\langle e \mid \mu \rangle \in \mathcal{R}_{L \rightarrow G\langle m \rangle \rightarrow G\langle m \rangle}$, then $\langle \text{fold}^M e \mid \mu \rangle \in \mathcal{R}_{G\langle n \rangle \rightarrow G\langle m \rangle}$.*

Proof (Sketch). We will prove that $\langle \text{fold}^M e' \mid \mu \cup \mu' \rangle \in \mathcal{R}_{G\langle m \rangle}$ holds for any $\langle e' \mid \mu' \rangle \in \mathcal{R}_{G\langle n \rangle}$ in three steps:

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

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

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

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

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

For Step 3, we show that there exists some k_0 such that $\langle \text{elim}^\emptyset (\text{fold}_{(k)}^M e e') \mid \mu \cup \mu' \rangle$ terminates for some $k \geq k_0$ if and only if $\langle \text{elim}^\emptyset (\text{fold}^M e e') \mid \mu \cup \mu' \rangle$ terminates. Since we have $\langle e' \mid \mu' \rangle \in \mathcal{R}_{G\langle n \rangle}$, we have that $\langle \text{elim}^\emptyset e' \mid \mu' \rangle$ terminates. Then, we can show that v' that occurs as $\langle \text{elim}^\emptyset (\text{fold}_{(k)}^M e e') \mid \mu \cup \mu' \rangle \rightarrow^* \langle E[\text{fold}_{(k)}^M e v'] \mid _ \rangle$ also occurs as $\langle \text{elim}^M e' \mid \mu' \rangle \rightarrow^* \langle \text{elim}^{M'} v' \mid _ \rangle$. From the termination of $\langle \text{elim}^\emptyset e' \mid \mu' \rangle$, we can say that the number

of arguments of $\text{fold}_{(k)}^M e$ is at most finite. Thus, by memoization, let k_0 be the number of such v' 's, $\langle \text{elim}^\emptyset (\text{fold}_{(k)}^M e e') \mid \mu \cup \mu' \rangle$ terminates if and only if $\langle \text{elim}^\emptyset (\text{fold}^M e e') \mid \mu \cup \mu' \rangle$ does, for all $k > k_0$. Thus, we have $\langle \text{fold}^M e e' \mid \mu \cup \mu' \rangle \in \mathcal{R}_{G\langle m \rangle}$. \square

7.4 Proof of Termination

Now we are ready to prove Theorem 5. To prove the theorem, we prove the following more general property.

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

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

Case T-VARREC. In this case, we have $e = x \in \text{dom}(\mu)$. By the induction hypothesis, we have $\langle \mu(x) \mid \mu, x = \bullet \rangle \in \mathcal{R}_\tau$. By Lemma 6, we have $\langle x \mid \mu \rangle \in \mathcal{R}_\tau$. Then, by 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 $\tau = G\langle n \rangle$. Then, $\langle \text{elim}^\emptyset (e_1 : e_2) \mid \eta \rangle$ has the reduction sequence

$$\begin{aligned} \langle \text{elim}^\emptyset (e_1 : e_2) \mid \eta \rangle &\rightarrow^* \langle \text{elim}^\emptyset (x_1 : x_2) \mid x_1 = e_1, x_2 = e_2, \eta \rangle \\ &\rightarrow^* \langle E[a] \mid a = x_1, r = \text{elim}^M x_2, x_1 = e_1, x_2 = e_2, \eta \rangle \end{aligned}$$

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

$$\dots \rightarrow^* \langle \text{elim}^{M''} x_2 \mid x_1 = a, x_2 = e_2, \eta' \rangle$$

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

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 $G\langle n \rangle$ corresponds to a finite graph, in the sense that \rightarrow terminates under the observation elim^\emptyset . \square

If we include isEmpty^M , it must be treated similarly to fold^M to keep Lemma 4. That is, also Bool has generations and isEmpty^M increases the generation, while if keeps them.

8. Related Work

We have discussed transformations of graphs up to *bisimilarity* like UnCAL [8]. Many frameworks have been proposed for transformations of graphs up to *equality/isomorphism*, from the functional programming community [10, 13–15, 21, 29, 31] and from the database community [1, 9]. Due to the difference of graph models, these results are incomparable with UnCAL and ours. Leveraging 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 like UnCAL cannot compute them.

In the listed above, some frameworks [10, 15, 21] 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. We did not use μ -terms and adopted expressions with heaps because of treatment of tuples: While we want to identify $\pi_1 \mu x. (\pi_2 x, \pi_1 x)$ with the black hole for example, it is nontrivial

to obtain reduction rules that reduce $\pi_1 \mu x. (\pi_2 x, \pi_1 x)$ to the black hole. In contrast, $\pi_1 (\text{fix}_G (\lambda x. (\pi_2 x, \pi_1 x)))$ successfully evaluates to \bullet in [38] and ours. In addition, even with μ -terms, we have to exclude problematic examples such as $\mu x.b : \text{ins} A x$.

CoCaml has “recursions” with user-specified resolvers of their fixpoints [28]. For example, a memoized recursion to write some terminating transformations on cyclic data can be realized by the constructor resolver. However, they do not give any formal system to guarantee that such a memoized recursion on a cyclic data has the same meaning as the corresponding usual recursion on the infinite data, while we gave it by the type system.

Nishimura and Ohori [39] discuss recursions similar to **srec**, for application of parallel queries on object-oriented databases [40]. Surprisingly, their idea of computation is quite similar to the bulk semantics of UnCAL [8], while general paramorphisms are not supported in their framework.

Hamana and the authors [22] discuss a categorical and algebraic characterizations of full UnCAL transformations. In the present paper, leveraging this idea, we clarified relationship between UnCAL and a lazy language by prohibiting the argument of **srec** to have output markers, materialized the concrete operational semantics and designed type system for the regularity guarantee according to the relationship. We also showed the effectiveness our idea by experiments.

9. Conclusion

We formalized the translation from UnCAL programs to FUNCAL ones, and then designed the semantics and type system of FUNCAL to execute the translated programs as finite-graph transformations with termination guarantee. Since FUNCAL is a variant of a call-by-need λ -calculus with memoized folds, this translation makes UnCAL graph transformations more amenable to program-manipulation techniques such as verification and optimization. Extending our system to general cyclic data is a future direction.

Acknowledgments

We thank Meng Wang, Janis Voigtländer, Patrik Jansson, and Makoto Hamana for their helpful comments. This work was partially supported by JSPS KAKENHI Grant Numbers 15K15966, 15H02681, 25540001, 24700020 and 22800003, and the Grand-Challenging Project on the “Linguistic Foundation for Bidirectional Model Transformation” of the National Institute of Informatics.

References

- [1] Serge Abiteboul, Dallan Quass, Jason McHugh, Jennifer Widom, and Janet L. Wiener. The Lorel query language for semistructured data. *Int. J. on Digital Libraries*, 1(1):68–88, 1997.
- [2] Zena M. Ariola and Stefan Blom. Cyclic lambda calculi. In *TACS*, volume 1281 of *LNCS*, pages 77–106. Springer, 1997.
- [3] Zena M. Ariola and Jan Willem Klop. Equational term graph rewriting. *Fundam. Inform.*, 26(3/4):207–240, 1996.
- [4] François Bancilhon and Nicolas Spyratos. Update semantics of relational views. *ACM Trans. Database Syst.*, 6(4):557–575, 1981.
- [5] Jean Bézivin, Bernhard Rumpe, Andy Schürr, and Laurence Tratt. Model transformations in practice workshop. In *MoDELS Satellite Events*, volume 3844 of *LNCS*, pages 120–127. Springer, 2005.
- [6] Peter Buneman, Susan Davidson, Gerd Hillebrand, and Dan Suciu. A query language and optimization techniques for unstructured data. In *SIGMOD*, pages 505–516. ACM, 1996.
- [7] Peter Buneman, Susan B. Davidson, Mary F. Fernandez, and Dan Suciu. Adding structure to unstructured data. In *ICDT*, volume 1186 of *LNCS*, pages 336–350. Springer, 1997.
- [8] Peter Buneman, Mary F. Fernandez, and Dan Suciu. UnQL: A query language and algebra for semistructured data based on structural recursion. *VLDB J.*, 9(1):76–110, 2000.
- [9] Mariano P. Consens and Alberto O. Mendelzon. GraphLog: a visual formalism for real life recursion. In *PODS*, pages 404–416. ACM Press, 1990.
- [10] Bruno C. d. S. Oliveira and William R. Cook. Functional programming with structured graphs. In *ICFP*, pages 77–88. ACM, 2012.
- [11] Umeshwar Dayal and Philip A. Bernstein. On the correct translation of update operations on relational views. *ACM Trans. Database Syst.*, 7(3):381–416, 1982.
- [12] Agostino Dovier, Carla Piazza, and Alberto Policriti. An efficient algorithm for computing bisimulation equivalence. *Theoretical Computer Science*, 311(1):221 – 256, 2004.
- [13] Martin 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.
- [14] Martin Erwig. Inductive graphs and functional graph algorithms. *J. Funct. Program.*, 11(5):467–492, 2001.
- [15] Leonidas Fegaras and Tim Sheard. Revisiting catamorphisms over datatypes with embedded functions (or, programs from outer space). In *POPL*, pages 284–294, 1996.
- [16] Sebastian Fischer, Oleg Kiselyov, and Chung-chieh Shan. Purely functional lazy non-deterministic programming. In *ICFP*, pages 11–22, 2009.
- [17] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Trans. Program. Lang. Syst.*, 29(3), 2007.
- [18] Nate Foster, Kazutaka Matsuda, and Janis Voigtländer. Three complementary approaches to bidirectional programming. In Jeremy Gibbons, editor, *SSGIP*, volume 7470 of *Lecture Notes in Computer Science*, pages 1–46. Springer, 2010. ISBN 978-3-642-32201-3.
- [19] Andrew J. Gill, John Launchbury, and Simon L. Peyton Jones. A short cut to deforestation. In *FPCA*, pages 223–232, 1993.
- [20] Lars Grunske, Leif Geiger, and Michael Lawley. A graphical specification of model transformations with triple graph grammars. In *ECMDA-FA*, volume 3748 of *LNCS*, pages 284–298. Springer, 2005.
- [21] Makoto Hamana. Initial algebra semantics for cyclic sharing tree structures. *Logical Methods in Computer Science*, 6(3), 2010.
- [22] Makoto Hamana, Kazutaka Matsuda, and Kazuyuki Asada. The algebra of recursive graph transformation language UnCAL: Complete axiomatisation and iteration categorical semantics. *Math. Struct. in Comp. Science*, accepted.
- [23] Soichiro Hidaka, Zhenjiang Hu, Kazuhiro Inaba, Hiroyuki Kato, Kazutaka Matsuda, and Keisuke Nakano. Bidirectionalizing graph transformations. In *ICFP*, pages 205–216. ACM, 2010.
- [24] Soichiro Hidaka, Zhenjiang Hu, Kazuhiro Inaba, Hiroyuki Kato, Kazutaka Matsuda, Keisuke Nakano, and Isao Sasano. Marker-directed optimization of UnCAL graph transformations. In *LOPSTR*, volume 7225 of *LNCS*, pages 123–138. Springer, 2011.
- [25] Soichiro Hidaka, Kazuyuki Asada, Zhenjiang Hu, Hiroyuki Kato, and Keisuke Nakano. Structural recursion for querying ordered graphs. In *ICFP*, pages 305–318, 2013.
- [26] Soichiro Hidaka, Zhenjiang Hu, Kazuhiro Inaba, Hiroyuki Kato, and Keisuke Nakano. GRoundTram: An integrated framework for developing well-behaved bidirectional model transformations. *Progress in Informatics*, (10):131–148, 2013.
- [27] Kazuhiro Inaba, Soichiro Hidaka, Zhenjiang Hu, Hiroyuki Kato, and Keisuke Nakano. Graph-transformation verification using monadic second-order logic. In *PPDP*, pages 17–28. ACM, 2011.
- [28] Jean-Baptiste Jeannin, Dexter Kozen, and Alexandra Silva. CoCaml: Programming with coinductive types. <http://www.cs.cornell.edu/~kozen/Papers/cocaml.pdf>, 2013.
- [29] Thomas Johnsson. Efficient graph algorithms using lazy monolithic arrays. *J. Funct. Program.*, 8(4):323–333, 1998.

- [30] Frédéric Jouault and Jean Bézivin. KM3: A DSL for metamodel specification. In *FMOODS*, volume 4037 of *LNCS*, pages 171–185. Springer, 2006.
- [31] David J. King and John Launchbury. Structuring depth-first search algorithms in haskell. In *POPL*, pages 344–354, 1995.
- [32] John Launchbury. A natural semantics for lazy evaluation. In *POPL*, pages 144–154, 1993.
- [33] Kazutaka Matsuda and Meng Wang. Bidirectionalization for free with runtime recording: or, a light-weight approach to the view-update problem. In Ricardo Peña and Tom Schrijvers, editors, *PPDP*, pages 297–308. ACM, 2013. ISBN 978-1-4503-2154-9.
- [34] Kazutaka Matsuda and Meng 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.
- [35] Lambert G. L. T. Meertens. Paramorphisms. *Formal Asp. Comput.*, 4(5):413–424, 1992.
- [36] Erik Meijer, Maarten M. Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *FPCA*, volume 523 of *LNCS*, pages 124–144. Springer, 1991.
- [37] Robin Milner. *Communicating and mobile systems - the Pi-calculus*. Cambridge University Press, 1999. ISBN 978-0-521-65869-0.
- [38] Keiko Nakata and Masahito Hasegawa. Small-step and big-step semantics for call-by-need. *J. Funct. Program.*, 19(6):699–722, 2009.
- [39] Susumu Nishimura and Atsushi Ohori. Parallel functional programming on recursively defined data via data-parallel recursion. *J. Funct. Program.*, 9(4):427–462, 1999.
- [40] Susumu Nishimura, Atsushi Ohori, and Keishi Tajima. An equational object-oriented data model and its data-parallel query language. In *OOPSLA*, pages 1–17, 1996. doi: 10.1145/236337.236339.
- [41] Atsushi Ohori and Isao Sasano. Lightweight fusion by fixed point promotion. In *POPL*, pages 143–154. ACM, 2007.
- [42] Yannis Papakonstantinou, Hector Garcia-Molina, and Jennifer Widom. Object exchange across heterogeneous information sources. In *ICDE*, pages 251–260. IEEE Computer Society, 1995.
- [43] Morten Heine Sørensen, Robert Glück, and Neil D. Jones. A positive supercompiler. *J. Funct. Program.*, 6(6):811–838, 1996.
- [44] William W. Tait. Intensional interpretations of functionals of finite type I. *Journal of Symbolic Logic*, 32(2):198–212, June 1967.
- [45] Hiroshi Unno, Naoshi Tabuchi, and Naoki Kobayashi. Verification of tree-processing programs via higher-order model checking. In *APLAS*, volume 6461 of *LNCS*, pages 312–327. Springer, 2010.
- [46] Janis Voigtländer. Bidirectionalization for free! (pearl). In Zhong Shao and Benjamin C. Pierce, editors, *POPL*, pages 165–176. ACM, 2009. ISBN 978-1-60558-379-2.
- [47] Yingfei Xiong, Dongxi Liu, Zhenjiang Hu, Haiyan Zhao, Masato Takeichi, and Hong Mei. Towards automatic model synchronization from model transformations. In R. E. Kurt Stirewalt, Alexander Egyed, and Bernd Fischer, editors, *ASE*, pages 164–173. ACM, 2007. ISBN 978-1-59593-882-4.
- [48] Yijun Yu, Yu Lin, Zhenjiang Hu, Soichiro Hidaka, Hiroyuki Kato, and Lionel Montrieux. Maintaining invariant traceability through bidirectional transformations. In *ICSE*, pages 540–550, 2012.

Proofs

The supplementary 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 $\Delta, x : G\langle n \rangle \vdash e : G\langle n \rangle$. Then, if $e \Rightarrow^* E[x]$ for some call-by-name evaluation context E (i.e., $E ::= \square \mid E e \mid (\text{fold } n e) E$), E must be the hole.*

Proof. It is easy to see the type system has the preservation property, and thus $\Delta \vdash E[x] : G\langle n \rangle$. Only the typable context is \square . \square

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 $\Gamma \vdash g : \text{DB}_Y^X \rightsquigarrow e$. Let Δ be a typing environment satisfying that $\Delta \vdash e :: (G\langle n \rangle)^{|Y|} \rightarrow (G\langle n \rangle)^{|X|}$ holds and that, for any x , $\Gamma(x) = \text{DB}_Y^X$, implies $\Delta(x) = (G\langle m \rangle)^{|Y'|} \rightarrow (G\langle m \rangle)^{|X'|}$. Let θ and η be substitutions that have the same domain and satisfy that, for any x in the domain, $\llbracket \theta(x) \rrbracket \approx \eta(x)$, $\vdash \theta(x) :: \Gamma(x)$ and $\vdash \eta(x) :: \Delta(x)$ hold. Then, we have $\llbracket g\theta \rrbracket \approx \eta$.*

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

Proof. We prove the theorem by the induction on the derivation tree of a conversion judgment $\Gamma \vdash g : \text{DB}_Y^X \rightsquigarrow e$. 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

$$\llbracket (g_1 @ g_2)\theta \rrbracket \approx \lambda y. e_1(e_2 y)\eta$$

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

$$\llbracket (g_1 @ g_2)\theta \rrbracket @ G_3 \sim (e'_1 (e'_2 e_3))$$

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

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

Case: C-CYC We will prove

$$\llbracket \text{cycle}(g)\theta \rrbracket \approx \lambda \bar{Y}. \text{fix}_G(\lambda \bar{X}. e \bar{X} \bar{Y} \bar{Y})\eta$$

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

$$\begin{aligned} & \llbracket \text{cycle}(g)\theta \rrbracket @ G_2 \\ & \sim \text{fix}_G(\lambda x. e(\{x\} \cup Y[\pi_1 e_2/y_1, \dots, \pi_m e_2/y_m])\eta) \end{aligned}$$

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 $(\llbracket \text{cycle}(g)\theta \rrbracket @ G_2)$, and G_1 be $\llbracket g\theta \rrbracket$. Then, G and $G_1 @ G_2$ only differ in the nodes that have the output marker $\&$.

That is, a node that has the output marker $\&x$ in $G_1 @ G_2$ 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 $G_1 @ G_2$. We write $\lambda x. C_0[x]$ for $\lambda x. e'(\{x\} \cup Y[\pi_1 e'_2/y_1, \dots, \pi_m e'_2/y_m])$. Then, we have $G_1 @ G_2 \approx \lambda x. C_0[x]$ from the induction hypothesis.

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

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

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

Suppose that we have $(v, e'') \in \mathcal{X}'$ and $v \rightarrow^\varepsilon \dots \rightarrow^\varepsilon v'$. 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 $(v, C[0 : \Omega]) \in \mathcal{X}$. We can safely ignore Case (1-i) because we have $e_0 \rightarrow^* C[e_0]$ and $(v_0, C[0 : \Omega]) \in \mathcal{X}$; the proof is covered by the second case. For Case (1-ii), where $e'' = C[e_0]$ with $(v, C[0 : \Omega]) \in \mathcal{X}$, there is some C' such that $(v', C'[0 : \Omega]) \in \mathcal{X}$ and $C[0 : \Omega] \Rightarrow^* a : C'[0 : \Omega]$ from Lemma 10 (note that we have $x :: G \vdash C[x] :: G$, which can be shown by the preservation property and $x :: G \vdash C_0[x] :: G$). Then, we have $C[e_0] \Rightarrow^* a : C'[e_0]$ 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 $u \rightarrow^\varepsilon u'$ that is not in G' . Then, u' must be v_0 from the evaluation of cycle and $u \rightarrow^0 v_H$ in G' . Thus, there must be the reduction $C[0 : \Omega] \Rightarrow^* C'[0 : \Omega] \Rightarrow^* 0 : \Omega$ 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 $e_0 \rightarrow^* C_0[e_0]$. Since the node v_0 is also the root of the graph G' , we have $(v_0, C_0[0 : \Omega]) \in \mathcal{X}$ 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 $e'' \Rightarrow^* a : e'''$. Similarly, there are three cases: (1) there is a corresponding evaluation sequence $C_1[0 : \Omega] \Rightarrow C_2[0 : \Omega] \Rightarrow \dots \Rightarrow a : C_n[0 : \Omega]$ 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 $e_0 \rightarrow^* C_0[e_0]$ (recall that \rightarrow is the call-by-name reduction) and we have $(v_0, C_0[e_0]) \in \mathcal{X}'$ from $(v_0, C_0[0 : \Omega]) \in \mathcal{X}$. For Case (3), since $C[e_0]$ and $C[0 : \Omega]$ must have the corresponding reduction unless $C = \square$, which is obtained from Lemma 10, the evaluation sequence must has the form of $e'' \Rightarrow^* e_0 \Rightarrow^* C[e_0] \Rightarrow^* a : e'''$, while the corresponding evaluation sequence is $C[0 : \Omega] \Rightarrow^* 0 : \Omega$. Then, we have $v \rightarrow^\varepsilon \dots \rightarrow^\varepsilon v_H \rightarrow^0 v'$ in G' . Note that v_H is the only node having an 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 $e'' \Rightarrow^* a : e'''$ is finite.

Case: C-REC We will prove

$$\begin{aligned} & \llbracket \text{srec}(\lambda(l, t). g_1)(g_2)\theta \rrbracket \\ & \approx \lambda y. \text{para } |Z| (\lambda l. \lambda t. (\lambda t. e_1) (\lambda_. t)) (e_2 ())\eta \end{aligned}$$

where $\Gamma, l :: L, t :: \text{DB}_\emptyset^{\{\&\}} \vdash g_1 : \text{DB}_Z^Z \rightsquigarrow e_1$ and $\Gamma \vdash g_2 : \text{DB}_\emptyset^{\{\&\}} \rightsquigarrow e_2$. 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

$$\llbracket \text{srec}(\lambda(l, t).g'_1)(g'_2) \rrbracket \sim \text{para } 1 (\lambda l. \lambda t'. (\lambda t. e'_1) (\lambda(). t')) (e'_2())$$

to prove the above. For simplicity, we directly handle **para** instead of fold. We write P for the expression $\text{para } 1 (\lambda l. \lambda t'. (\lambda t. e'_1) (\lambda(). t'))$.

Let G_0 and G_2 be graphs $\llbracket \text{srec}(\lambda(l, t).g'_1)(g'_2) \rrbracket$ and $\llbracket g'_2 \rrbracket$, 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 0_ζ , and r_ζ be its root. From the induction hypothesis, we have $\llbracket g_1[a/l, G/t] \rrbracket @ (\&z \triangleright H_\zeta) \sim e'_1[a/l, e/t] (0_\zeta : \Omega)$ for any G, e with $G \approx e$, where 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 **srec** in Section 3.

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

- $(v, C[P e'_2]) \in \mathcal{X}'$ if $v \in V_\zeta$, $(v, C[0_\zeta : \Omega]) \in \mathcal{X}_{\zeta, G, e}$ 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 $v \rightarrow^\varepsilon \dots \rightarrow^\varepsilon \rightarrow^a v'$. 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 $v \rightarrow^\varepsilon v''$ in the sequence and $e \rightarrow^* e''$, 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 $v \rightarrow^\varepsilon \dots \rightarrow^\varepsilon v'' \rightarrow^\varepsilon u_{v_2} \rightarrow^\varepsilon v''' \rightarrow^\varepsilon \dots$ 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 $(v, C[0_\zeta : \Omega]) \in \mathcal{X}_{\zeta, G', e'}$ for some G' and e' . Since $v'' \rightarrow^{0_\zeta} r_\zeta$, there is a reduction sequence $C[0_\zeta : \Omega] \Rightarrow^* 0_\zeta : \Omega$. Thus, there must also be a reduction sequence $C[P e'_2] \Rightarrow^* P e'_2$. Since $\zeta = (_, _, v_2)$ from the semantics of **srec** and $(v_2, e'_2) \in \mathcal{X}_2$, we have $(u_{v_2}, P e'_2) \in \mathcal{X}'$. 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. \square

B. Proof of Theorem 3

The basic idea is to assign an integer to an expression so that an expression that occurs as an argument of **srec** has a greater number than outer expressions. Formally, we prove the theorem by extending $\Gamma \vdash \text{DB}_Y^X : g \rightsquigarrow e$ to the 6-ary relation $\Gamma \vdash \text{DB}_Y^X : g \rightsquigarrow e : \tau, \Delta$ that reads an UnCAL expression g that has type DB_Y^X 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:

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

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 DB_0^* ; accordingly, Δ maps a variable x to type $() \rightarrow G\langle n \rangle$ for some n . Note that this property is closed under \sqcup . The rules for graph constructors basically do not touch

generation. The only rule that uses generation is C-REC. The rule C-REC says that the argument of **srec**($\lambda(l, t).g_1$) must be more traversal than its return type and t . It is worth noting that every τ of $\Gamma \vdash \text{DB}_Y^X : g \rightsquigarrow e : \tau, \Delta$ in Figure 9 has the form of $(G\langle n \rangle)^{|Y|} \rightarrow (G\langle n \rangle)^{|Z|}$ 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

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

and extend the typing rules for **fix_G** and **fold_e** to return tuples of graphs as below.

$$\frac{\Gamma \vdash e : (G\langle k_1 \rangle \times \dots \times G\langle k_n \rangle) \rightarrow (G\langle k_1 \rangle \times \dots \times G\langle k_n \rangle)}{\Gamma \vdash \text{fix}_G e : G\langle k_1 \rangle \times \dots \times G\langle k_n \rangle} \text{T-FIX}$$

$$\frac{\Gamma \vdash e : L \rightarrow (G\langle m_1 \rangle \times \dots \times G\langle m_n \rangle) \rightarrow (G\langle m_1 \rangle \times \dots \times G\langle m_n \rangle) \quad k < \min \{m_1, \dots, m_n\}}{\Gamma \vdash \text{fold}_n e : G\langle k \rangle \rightarrow (G\langle m_1 \rangle \times \dots \times G\langle m_n \rangle)} \text{T-CATA}$$

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 **para_n** e must obey the following typing rule.

$$\frac{\Gamma \vdash e : L \rightarrow G\langle k+1 \rangle \rightarrow (G\langle m_1 \rangle \times \dots \times G\langle m_n \rangle) \rightarrow (G\langle m_1 \rangle \times \dots \times G\langle m_n \rangle) \quad k < \min \{m_1, \dots, m_n\}}{\Gamma \vdash \text{para}_n e : G\langle k \rangle \rightarrow (G\langle m_1 \rangle \times \dots \times G\langle m_n \rangle)} \text{T-PARA}$$

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

- Suppose $\Gamma \vdash g : \text{DB}_Y^X \rightsquigarrow e : \tau, \Delta$, then $\Delta \vdash e :: \tau$.
- Suppose $\Gamma \vdash g : \text{DB}_Y^X \rightsquigarrow e$, then $\Gamma \vdash g : \text{DB}_Y^X \rightsquigarrow e : \tau, \Delta$ 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 **srec** holds if the corresponding premise of the original conversion rule for **srec** holds. This is done by proving the following more general property.

Lemma 11. Suppose $\Gamma \vdash g : \text{DB}_Y^X \rightsquigarrow e$, then, for any integer n , there is Δ such that $\Gamma \vdash g : \text{DB}_Y^X \rightsquigarrow e : (G\langle n \rangle)^{|Y|} \rightarrow (G\langle n \rangle)^{|X|}, \Delta$ and, for any $x \in \text{dom}(\Delta)$, $\Delta(x) = () \rightarrow G\langle m \rangle$ with $m \leq n$ holds. \square

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.

- $(e_1, e_2) \in \mathcal{B}_L$ iff $e_1 \rightarrow^* a$ and $e_2 \rightarrow^* a$ for some a ,
- $(e_1, e_2) \in \mathcal{B}_G$ iff $G_{e_1} \sim G_{e_2}$,
- $(f_1, f_2) \in \mathcal{B}_{\tau_1 \rightarrow \tau_2}$ iff $(f_1 e_1, f_2 e_2) \in \mathcal{B}_{\tau_2}$ for any $(e_1, e_2) \in \mathcal{B}_{\tau_1}$

Intuitively, \mathcal{B}_τ 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 $x \in \text{dom}(\Gamma)$.

$\frac{}{\Gamma \vdash \{\} : \text{DB}_Y^{\{\&\}} \rightsquigarrow \bullet : () \rightarrow \mathbf{G}\langle n \rangle, \emptyset} \text{C-SINGLE}$	
$\frac{\Gamma \vdash g : \text{DB}_Y^{\{\&\}} \rightsquigarrow e : (\mathbf{G}\langle n \rangle)^{ Y } \rightarrow \mathbf{G}\langle n \rangle, \Delta}{\Gamma \vdash \{l : g\} : \text{DB}_Y^{\{\&\}} \rightsquigarrow (\lambda y. l : e) : (\mathbf{G}\langle n \rangle)^{ Y } \rightarrow \mathbf{G}\langle n \rangle, \Delta} \text{C-EDGE}$	
$\frac{\left\{ \Gamma \vdash g_i : \text{DB}_Y^{\&} \rightsquigarrow e_i : (\mathbf{G}\langle n \rangle)^{ Y } \rightarrow \mathbf{G}\langle n \rangle, \Delta_i \right\}_{i=1,2}}{\Gamma \vdash g_1 \cup g_2 : \text{DB}_Y^{\&} \rightsquigarrow (\lambda y. (e_1 y) \cup (e_2 y)) : (\mathbf{G}\langle n \rangle)^{ Y } \rightarrow \mathbf{G}\langle n \rangle, \Delta_1 \sqcup \Delta_2} \text{C-UNI}$	
$\frac{\Gamma \vdash g : \text{DB}_Y^{\{\&\}} \rightsquigarrow e : (\mathbf{G}\langle n \rangle)^{ Y } \rightarrow \mathbf{G}\langle n \rangle, \Delta}{\Gamma \vdash \&x \triangleright g : \text{DB}_Y^{\{\&x\}} \rightsquigarrow e : (\mathbf{G}\langle n \rangle)^{ Y } \rightarrow \mathbf{G}\langle n \rangle, \Delta} \text{C-ROOT}$	
$\frac{y = y_i \text{ of } (y_1, \dots, y_n) = \bar{Y}}{\Gamma \vdash \&y : \text{DB}_Y^{\{\&\}} \rightsquigarrow \lambda y. \pi_i y : (\mathbf{G}\langle n \rangle)^{ Y } \rightarrow \mathbf{G}\langle n \rangle, \emptyset} \text{C-HOLE}$	
$\frac{}{\Gamma \vdash () : \text{DB}_Y^{\emptyset} \rightsquigarrow \lambda y. () : (\mathbf{G}\langle n \rangle)^{ Y } \rightarrow (), \emptyset} \text{C-EMP}$	
$\frac{\left\{ \Gamma \vdash g_i : \text{DB}_Y^{X_i} \rightsquigarrow e_i : (\mathbf{G}\langle n \rangle)^{ Y } \rightarrow (\mathbf{G}\langle n \rangle)^{ X_i }, \Delta_i \right\}_{i=1,2} \quad p \equiv \lambda \bar{X}_1. \lambda \bar{X}_2. \bar{X}_1 \cup \bar{X}_2}{\Gamma \vdash g_1 \oplus g_2 : \text{DB}_Y^{X_1 \sqcup X_2} \rightsquigarrow \lambda y. p (e_1 y) (e_2 y) : (\mathbf{G}\langle n \rangle)^{ Y } \rightarrow (\mathbf{G}\langle n \rangle)^{ X_1 \cup X_2 }, \Delta_1 \sqcup \Delta_2} \text{C-RU}$	
$\frac{\Gamma \vdash g_1 : \text{DB}_Y^X \rightsquigarrow e_1 : (\mathbf{G}\langle n \rangle)^{ Y } \rightarrow (\mathbf{G}\langle n \rangle)^{ X }, \Delta_1 \quad \Gamma \vdash g_2 : \text{DB}_Z^Y \rightsquigarrow e_2 : (\mathbf{G}\langle n \rangle)^{ Z } \rightarrow (\mathbf{G}\langle n \rangle)^{ Y }, \Delta_2}{\Gamma \vdash g_1 @ g_2 : \text{DB}_Z^X \rightsquigarrow \lambda y. e_1 (e_2 y) : (\mathbf{G}\langle n \rangle)^{ Z } \rightarrow (\mathbf{G}\langle n \rangle)^{ X }, \Delta_1 \sqcup \Delta_2} \text{C-SUBST}$	
$\frac{\Gamma \vdash g : \text{DB}_{X \sqcup Y}^X \rightsquigarrow e : (\mathbf{G}\langle n \rangle)^{ X \sqcup Y } \rightarrow (\mathbf{G}\langle n \rangle)^{ X }, \Delta}{\Gamma \vdash \text{cycle}(g) : \text{DB}_Y^X \rightsquigarrow \lambda \bar{Y}. \text{fix}_{\mathbf{G}}(\bar{X}. e \bar{X} \cup \bar{Y}) : (\mathbf{G}\langle n \rangle)^{ Y } \rightarrow (\mathbf{G}\langle n \rangle)^{ X }, \Delta} \text{C-CYC}$	
$\frac{}{\Gamma \vdash y : \Gamma(y) \rightsquigarrow y : () \rightarrow \mathbf{G}\langle n \rangle, \{y : \mathbf{G}\langle n \rangle\}} \text{C-VAR}$	
$\frac{\Gamma \vdash g : \text{DB}_Y^X \rightsquigarrow e : (\mathbf{G}\langle n \rangle)^{ Y } \rightarrow (\mathbf{G}\langle n \rangle)^{ X }, \Delta \quad Y \subseteq Y'}{\Gamma \vdash g : \text{DB}_{Y'}^X \rightsquigarrow \lambda \bar{Y}'. e \bar{Y}' : (\mathbf{G}\langle n \rangle)^{ Y' } \rightarrow (\mathbf{G}\langle n \rangle)^{ X }, \Delta} \text{C-SUB}$	
$\frac{\Gamma, l : \mathbf{L}, t : \text{DB}_{\emptyset}^{\{\&\}} \vdash g_1 : \text{DB}_Z^Z \rightsquigarrow e_1 : (\mathbf{G}\langle m \rangle)^{ Z } \rightarrow (\mathbf{G}\langle m \rangle)^{ Z }, \Delta_1 \quad \Delta_1(t) = () \rightarrow \mathbf{G}\langle m' \rangle}{\Gamma \vdash g_2 : \text{DB}_{\emptyset}^{\{\&\}} \rightsquigarrow e_2 : () \rightarrow (\mathbf{G}\langle \min\{m, m'\} - 1 \rangle), \Delta_2} \text{C-REC}$	
$\Gamma \vdash \text{srec}(\lambda(l, t). g_1)(g_2) : \text{DB}_{\emptyset}^Z \rightsquigarrow \lambda y. \text{para}_{ Z }(\lambda l. \lambda t'. (\lambda t. e_1)(\lambda(). t'))(e_2()) : () \rightarrow (\mathbf{G}\langle m \rangle)^{ Z }, \Delta_1 \sqcup \Delta_2$	

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 $\text{fix}_{\mathbf{G}} e\theta_1 \sim \text{fix}_{\mathbf{G}} e\theta_2$. Let us write e_1 for $e\theta_1$ and e_2 for $e\theta_2$. Then we have $(e_1, e_2) \in \mathcal{B}_{\mathcal{G} \rightarrow \mathcal{G}}$ 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.

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

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

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

- $\text{fix}_{\mathbf{G}} e_1 \Rightarrow e_1(\text{fix}_{\mathbf{G}} e_1) \Rightarrow^* C_1[\text{fix}_{\mathbf{G}} e_1] \Rightarrow^* a : C'_1[\text{fix}_{\mathbf{G}} e_1]$ where $e_1 \Omega \Rightarrow^* C_1[\Omega] \Rightarrow^* a : C'_1[\Omega]$, or

- $\text{fix}_{\mathbf{G}} e_1 \Rightarrow e_1(\text{fix}_{\mathbf{G}} e_1) \Rightarrow^* C_1[\text{fix}_{\mathbf{G}} e_1] \Rightarrow^* \text{fix}_{\mathbf{G}} e_1$ where $e_1 \Omega \Rightarrow^* C_1[\Omega] \Rightarrow^* \Omega$.

Then, e' must be the form of $C'_1[\text{fix}_{\mathbf{G}} e_1]$ from the above discussion. Then, there must be a corresponding sequence from $\text{fix}_{\mathbf{G}} e_2$ such that $\text{fix}_{\mathbf{G}} e_2 \Rightarrow e_2(\text{fix}_{\mathbf{G}} e_2) \Rightarrow^* C_2[\text{fix}_{\mathbf{G}} e_2] \Rightarrow^* a : C'_2[\text{fix}_{\mathbf{G}} e_2]$ from the induction hypothesis. As a result, from the definition, we obtain $(C'_1[\text{fix}_{\mathbf{G}} e_1], C'_2[\text{fix}_{\mathbf{G}} e_2]) \in \mathcal{X}$.

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

- $C_1[\text{fix}_{\mathbf{G}} e_1] \Rightarrow^* a : C'_1[\text{fix}_{\mathbf{G}} e_1]$ where $C_1[\Omega] \Rightarrow^* a : C'_1[\Omega]$, or
- $C_1[\text{fix}_{\mathbf{G}} e_1] \Rightarrow^* \text{fix}_{\mathbf{G}} e_1$ where $C_1[\Omega] \Rightarrow^* \Omega$.

For the former case, there must be a corresponding sequence from $C_2[\text{fix}_{\mathbf{G}} e_2]$ such that $C_2[\text{fix}_{\mathbf{G}} e_2] \Rightarrow^* a : C'_2[\text{fix}_{\mathbf{G}} e_2]$ where $C_2[\Omega] \Rightarrow^* a : C'_2[\Omega]$. From the definition, we have $(C'_1[\text{fix}_{\mathbf{G}} e_1], C'_2[\text{fix}_{\mathbf{G}} e_2]) \in \mathcal{X}$. For the latter case, we repeat the discussion in the previous paragraph.

Case: T-CATA. We will prove $\text{fold}(f\theta_1)e_1 \sim \text{fold}(f\theta_2)e_2$ 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 $f_1 a \Omega \sim f_2 a \Omega$ for any a where Ω is a fresh free variable. Let us write \mathcal{X}'_a is the bisimulation relation to prove $f_1 a \Omega \sim f_2 a \Omega$, and \mathcal{X}'' is that to prove $e_1 \sim e_2$.

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

$$\begin{aligned} (\text{fold } f_1 e'_1, \text{fold } f_2 e'_2) \in \mathcal{X} & \text{ if } (e'_1, e'_2) \in \mathcal{X}'' \\ (C_1[\text{fold } f_1 e'_1], C_2[\text{fold } f_2 e'_2]) \in \mathcal{X} & \text{ if } (C_1[\Omega], C_2[\Omega]) \in \mathcal{X}'_a \text{ and } (e'_1, e'_2) \in \mathcal{X}'' \text{ for some } a \end{aligned}$$

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

Assume that $\text{fold } f_1 e'_1 \Rightarrow^* a : e''$. Let us consider how $\text{fold } f_1 e'_1$ is reduced. From Lemma 10, there are only the two possibilities:

- $\text{fold } f_1 e'_1 \Rightarrow^* \text{fold } f_1 (c : e''_1) \Rightarrow^* f_1 a (\text{fold } f_1 e''_1) \Rightarrow^* C_1[\text{fold } f_1 e''_1] \Rightarrow^* b : C'_1[\text{fold } f_1 e''_1]$ where $e'_1 \Rightarrow a : e''_1$ and $f_1 a \Omega \Rightarrow^* C_1[\Omega] \Rightarrow^* b : C'_1[\Omega]$, or
- $\text{fold } f_1 e'_1 \Rightarrow^* \text{fold } f_1 (c : e''_1) \Rightarrow^* f_1 a (\text{fold } f_1 e''_1) \Rightarrow^* C_1[\text{fold } f_1 e''_1] \Rightarrow^* \text{fold } f_1 e''_1$ where $e'_1 \Rightarrow a : e''_1$ and $f_2 a \Omega \Rightarrow^* C_1[\Omega] \Rightarrow^* \Omega$.

In either case, $\text{fold } f_2 e'_2$ has the corresponding reduction from the induction hypothesis. Thus, it must be the case that $e'' = C_1[\text{fold } f_1 e''_1]$ and $\text{fold } f_2 e'_2$ has the corresponding reduction $\text{fold } f_2 e'_2 \Rightarrow^* C_2[\text{fold } f_1 e''_2]$ satisfying $(e''_1, e''_2) \in \mathcal{X}''$ and $(C_1[\Omega], C_2[\Omega]) \in \mathcal{X}'_a$ for some d . Then, from the definition, we obtain $(C_1[\text{fold } f_1 e''_1], C_2[\text{fold } f_1 e''_2]) \in \mathcal{X}$.

Assume that $C_1[\text{fold } f_1 e'_1] \Rightarrow^* a : e''$. From Lemma 10, there are only two possibilities about how $C_1[\text{fold } f_1 e'_1]$ is reduced:

- $C_1[\text{fold } f_1 e'_1] \Rightarrow^* b : C'_1[\text{fold } f_1 e'_1]$ where $e'_1 \Rightarrow a : e''_1$ and $C_1[\Omega] \Rightarrow^* b : C'_1[\Omega]$, or
- $C_1[\text{fold } f_1 e'_1] \Rightarrow^* \text{fold } f_1 e'_1$ where $e'_1 \Rightarrow a : e''_1$ and $C_1[\Omega] \Rightarrow^* \Omega$.

In either case, $C_2[\text{fold } f_2 e'_2]$ 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 $\langle \text{elim}^M e \mid \mu \rangle$. 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 $c\sigma_1 = \langle \text{elim}^M e\sigma_1 \mid \mu\sigma_1 \rangle$ and $c\sigma_2 = \langle \text{elim}^M e\sigma_2 \mid \mu\sigma_2 \rangle$. 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 $\langle \text{elim}^{M'} z_i\sigma_1 \mid \mu'\sigma_1 \rangle$ and $\langle \text{elim}^{M'} z_i\sigma_2 \mid \mu'\sigma_2 \rangle$ 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 $z_i\sigma_j \in \text{dom}(M')$ or $z_i\sigma_j = \bullet$ holds for $j = 1, 2$. \square