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 [16], and Object Exchange Model (OEM) for exchanging arbitrary database structures [36]. In such circumstances, several languages, such as UnQL/UnCAL [7], Lorel [1], and GraphLog [8], 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.

UnCAL is a prominent language designed for graph transformations [7]. Among its other nice features such as termination guarantee and efficient execution by ε-edges [7], 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 [10]. 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 [23] as well as optimization [7, 20].

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 [7, 20] is quite similar to the classic fold-fusion [30].

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.

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.

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

PEPM'17, January 16–17, 2017, Paris, France
© 2017 ACM. 978-1-4503-4721-1/17/01...$15.00
http://dx.doi.org/10.1145/3018882.3018883

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

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

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

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

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

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

we get the following graph.

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

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

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

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:

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

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

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

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.

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

The one also would have noticed that the behavior of **srec** could be expressed by a paramorphism [29] *para* that behaves like:

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

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

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

The transformation can be seen as a simplified version of the above transformation that searches Bob, in the sense that it models the behavior of the second-outermost **srec** of the transformation when it is applied to a graph with no 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 $\mathbf{cycle}(\&x \triangleright \{\texttt{a} : \&x\})$. The UnCAL expression

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

terminates and returns a singleton graph while the corresponding functional program

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

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

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

1.2 Contributions

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 [32] with the black hole [2, 3, 32] and memoization. This helps us to implement the semantics easily, which runs faster than the existing implementation of UnCAL [22] (Section 2.3). 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).

Due to the space limitation, we omit some proofs, which can be found in the full version available from `http://www2.sf.ecei.tohoku.ac.jp/~kztk/papers/funcal_full.pdf`.

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.

2.1 Verification

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

A few but interesting results are known for the verification problem on UnCAL. Buneman et al. [6] 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. [23] 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. [39], 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. [23]'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 [21], then Inaba et al. [23]'s method becomes no longer applicable. In contract, the method by Unno et al. [39] 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 [7, 20]. 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. [20], 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 contains 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 [30]. Since the expression e @ ($\&x \triangleright e'$) is converted to expression $(\lambda x.e)\ e'$ by our translation, we can regard Hidaka et al. [20]'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 [37], short-cut fusion [15] and lightweight fusion [35]. 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.

	$\lvert V\rvert$	$\lvert E\rvert$	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
`C2O_Sel`	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).

We implemented our semantics in Section 5 as an embedded DSL on Haskell;[1] inspired by [14], 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 [7] (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 [19, 20, 22][2], an UnCAL implementation using OCaml. GRoundTram implements two evaluation strategies: bulk semantics [7] that evaluates recursions as map-like operations on edges, and recursive semantics that uses memoized recursions to realize **srec**, both equipped with some optimizations [7, 20]. 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 [4], `PIM2PSM` (from [20]) converts a platform independent model to a platform specific model, and `C2O_Sel` (from [20]) converts a customer-order database from a customer-oriented representation to a order-oriented representation with some extraction, and `a2d_xc` (from [19]) renames `A` to `D` and contracts `C`. The program codes of `Class2RDB`, `PIM2PSM` and `C2O_Sel` are mechanically generated; they are originally written in UnQL$^+$ [22] and converted to UnCAL. For `a2d_xc`, we used the three graphs as input: `S30k` is a 30000-long sequence of $\searrow\!\circ\xrightarrow{A}\circ\xrightarrow{A}\cdots\xrightarrow{A}\circ$, `M200` is a lattice-like graph of 40000 nodes in which the i-th node connects to the $(i+1)$-th and $(i+200)$-th nodes modulo 40000 by `A`, and `C200` is a graph of 201 nodes in which every node connects to the other nodes by `A`. 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 `C2O_Sel` contain 57, 37 and 14 **srec**s, 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

[1] `https://bitbucket.org/kztk/funcal`

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

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 `C2O_Sel` 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.

3. Brief Overview of UnCAL

In this section, we briefly overview UnCAL [7], a first-order functional programming language that manipulates graphs. UnCAL is an internal language of UnQL [7] and UnQL$^+$ [22], 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
```

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 [19, 22, 40], 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 [7].

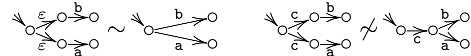
Formally, UnCAL graphs are defined as follows. Let $\mathcal{M}$ be a set of markers and L be a certain set of labels. An *UnCAL graph* G is a quadruple (V, E, I, O), where V is a set of nodes, $E \subseteq V \times (\mathsf{L} \cup \{\varepsilon\}) \times V$ is a set of edges, $I \subseteq \mathcal{M}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

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

Figure 1. The syntax of the positive subset of UnCAL

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 $v_1 \xrightarrow{\varepsilon}{}^* w_1 \xrightarrow{\mathtt{a}} u_1$ 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) $\mathsf{dom}(I_1) = \mathsf{dom}(I_2)$ and $(I_1(\&x), I_2(\&x)) \in \mathcal{X}$ for any $\&x \in \mathsf{dom}(I_1) = \mathsf{dom}(I_2)$. Two graphs G_1 and G_2 are called *bisimilar*, denoted by $G_1 \sim G_2$, if there is a bisimulation between G_1 and G_2.

Note that the graph bisimulation is different from weak bisimulation [31] 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 1 shows the positive [7] 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 [7], the subset does not contain **if**-expressions and the $\mathsf{isEmpty}$ 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 $\mathsf{isEmpty}$, as there is no computable counterpart of $\mathsf{isEmpty}$ 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 $\mathsf{isEmpty}$ because $\mathsf{isEmpty}$ 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 men-

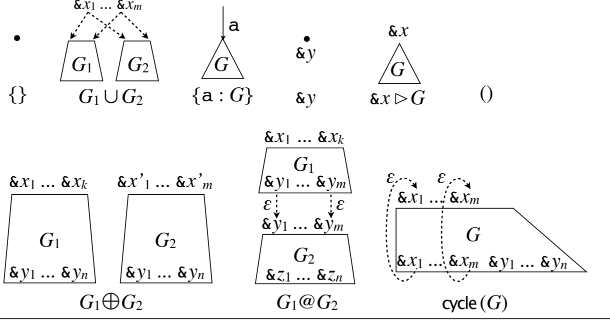


Figure 2. Graph Constructors

tioned in Section 1.1. A record notation shown in Section 1.1 such as $\{\mathtt{name} : \mathtt{Alice}, \mathtt{email} : \mathtt{alice}\}$ is a syntax sugar for $\{\mathtt{name} : \{\mathtt{Alice} : \{\}\}\} \cup \{\mathtt{email} : \{\mathtt{alice} : \{\}\}\}$. Figure 2 illustrates their intuitive behaviors. In what follows, we introduce the formal definitions of these nine constructors each by each.

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

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

Here v is a fresh node.

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

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

Here, u is a fresh node.

Edge-set Union The expression $g_1 \cup g_2$ adds two ε-edges from the new root to the roots of G_1 and G_2, where G_1 and G_2 are evaluation results of g_1 and g_2, respectively. Formally, its semantics is defined by:[3]

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

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

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

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

Here v is a fresh node.

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

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

Unlike the original definition [7], we restrict that the input markers of g must be the singleton $\{\&\}$ for simplicity.

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

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

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

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

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

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

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

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

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

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

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

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

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

Structural Recursion The expression $\mathbf{srec}(\lambda(l, t).g)(_)$ represents a *structural recursion* in the sense that a function $f(x) = \mathbf{srec}(\lambda(l, t).g)(x)$ satisfies the following laws [7].

$$\begin{aligned} f(\{\}) &= \{\} && \text{(SR1)} \\ f(\{\mathtt{a} : G\}) &= g[\mathtt{a}/l, G/t] \ @\ f(G) && \text{(SR2)} \\ f(G_1 \cup G_2) &= f(G_1) \cup f(G_2) && \text{(SR3)}\end{aligned}$$

Thanks to $\mathbf{srec}$, UnCAL can express many graph transformations in an efficient way, with guarantee of termination [7, 22].

Formally, its semantics (the bulk semantics [7]) is defined by:

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

Here, V' are fresh nodes, and $Z = \mathsf{dom}(I_\zeta)$ and $\mathsf{ran}(O_\zeta) \subseteq Z$ for each $\zeta \in E$. We assume that $\mathsf{dom}(I_\zeta) = \mathsf{dom}(I_{\zeta'})$ for all $\zeta, \zeta' \in E$, and V_ζ and $V_{\zeta'}$ are disjoint for different $\zeta, \zeta' \in E$. Intuitively, the semantics computes $g[\mathtt{a}/l, G/t]$ for each edge in $[\![g']\!]$, and connects them by ε-edges which corresponds to (SR2) and (SR3). Unlike the original definition [7], we require the graph $[\![g']\!]$ to have only one root named & and no holes. The former restriction is just for simplicity. For typed UnCAL [7], we can convert UnCAL programs to ones that satisfy the condition. In

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

Figure 3. The syntax of FUnCAL

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 [5, 20], 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 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 $\mathbf{cycle}(\&)$ satisfying

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

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

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

4.1 Translation

The syntax of the target language of our translation is given in Figure 3. 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 ith element from an n tuple), (infinite) tree constructors, and the (first-order) fixed-point operator $\mathsf{fix}_{\mathbf{G}}$, and structural recursions $\mathsf{fold}_n\ f$. Tree constructors consist of a leaf $\bullet$, edge extension $(:)$, and branch construction $\cup$. For simplicity, we shall write $a : b$ for $\{a : b\}$ henceforth. We use $\mathsf{fix}_{\mathbf{G}}$ instead of $\mathbf{letrec}$ that appeared in Section 1.1, since it is handy to discuss reductions. The structural recursion $\mathsf{fold}_n\ f$ is "fold" for the tree constructors defined recursively as:

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

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

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

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

where G is the (coinductive) datatype defined by:

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

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

We 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 $\overline{X}$ for a tuple $(x_1, \ldots, x_n)$ where $\{\&x_1, \ldots, \&x_n\} = X$ and $\&x_i < \&x_j$ $(i < j)$. Sometimes, we use a syntax sugar $\lambda\overline{X}.e$ for $\lambda t.e[(\pi_i t)/x_i]_{1 \le i \le n}$ where $(x_1, \ldots, x_n) = \overline{X}$. For example, assuming $x_1 < x_2$, we write $\lambda(x_1, x_2).x_1$ for $\lambda t.\pi_1 t$. 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 $\Gamma \vdash g : \mathsf{DB}_Y^X \rightsquigarrow e$ 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 $\mathsf{DB}_{Y'}^{X'}$ to $\mathsf{G}^{|Y'|} \to \mathsf{G}^{|X'|}$. Figure 4 shows the translation rules for the UnCAL graph constructors. If we ignore the $\rightsquigarrow e$ part of $\Gamma \vdash g : \mathsf{DB}_Y^X \rightsquigarrow e$, the judgment and rules coincide to the typing judgment and rules of UnCAL [5, 20].

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

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

Here, $\mathsf{para}_n e$, representing a "paramorphism"[4] [29], where an expression $\lambda f.\mathsf{para}_n f$ has type $(\mathsf{L} \to \mathsf{G} \to \mathsf{G}^n \to \mathsf{G}^n) \to \mathsf{G} \to \mathsf{G}^n$, is a syntax sugar defined by:

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

where $p : \mathsf{G}^{n+1} \to \mathsf{G}^n$, $p' : \mathsf{G}^n \to \mathsf{G}$ and $q : \mathsf{G}^n \to \mathsf{G} \to \mathsf{G}^{n+1}$ are functions to rearrange tuples defined as $p\ x = (\pi_1 x, \ldots, \pi_n x)$, $p'\ x = \pi_{n+1} x$, and $q\ x\ y = (\pi_1 x, \ldots, \pi_n x, y)$. This definition of para_n is similar to how a paramorphism is represented by a catamorphism (fold) via tupling [29]. The rule C-REC becomes a bit complicated due to explicit conversions between G and $() \to \mathsf{G}$. Since the argument of para_n must be of type G instead of $() \to \mathsf{G}$, we apply $()$ to e_2. In addition, since the conversion assumes that t in e_1 has type $() \to \mathsf{G}$, we construct such a function by $\lambda().t'$.

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

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

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

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

$$\frac{\Gamma \vdash g : \mathsf{DB}_Y^{\{\&\}} \rightsquigarrow e}{\Gamma \vdash \&x \rhd g : \mathsf{DB}_Y^{\{\&x\}} \rightsquigarrow e}\text{C-ROOT} \qquad \frac{}{\Gamma \vdash () : \mathsf{DB}_Y^{\emptyset} \rightsquigarrow \lambda y.()}\text{C-EMP}$$

$$\frac{\overline{Y} = (\&y_1, \ldots, \&y_n) \quad \&y = \&y_i \quad 1 \le i \le n}{\Gamma \vdash \&y : \mathsf{DB}_Y^{\{\&\}} \rightsquigarrow \lambda z.\pi_i z}\text{C-HOLE}$$

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

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

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

$$\frac{z\text{: graph/label variable}}{\Gamma \vdash z : \Gamma(z) \rightsquigarrow z}\text{C-VAR} \qquad \frac{\Gamma \vdash g : \mathsf{DB}_Y^X \rightsquigarrow e \quad Y \subseteq Y'}{\Gamma \vdash g : \mathsf{DB}_{Y'}^X \rightsquigarrow \lambda\overline{Y'}.e\ \overline{Y}}\text{C-SUB}$$

Figure 4. Conversion rules of UnCAL graph-constructors.

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

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* $\mathsf{G}^{|Y|} \to \mathsf{G}^{|X|}$ in FUnCAL rather than an expression that generates a (tuple of) tree. To leap the gap, we first define a relation between output-marker-free UnCAL graphs and FUnCAL tree expressions, and then we extend the relation to one that between general UnCAL graphs and FUnCAL functions.

First, we define a graph obtained from an expression as a labeled transition system [31].

Definition 1. A *reduction graph* $G_{e_0,X}$ of a (possibly-open) FUnCAL expression e_0 of type G^n and markers $X = \{\&x_1, \ldots, \&x_n\}$ 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, $E = \{(e, \mathsf{a}, e') \mid e \Rightarrow^* \mathsf{a} : e'\}$, and $I = \{\&x_1 \mapsto \pi_1 e_0, \ldots, \&x_n \mapsto \pi_n e_0\}$. Here, the relation $(\Rightarrow)$ is defined by:

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

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

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

Now, we have the following theorem.

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

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 [7] 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 [22].

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 $e \Rightarrow^* \mathsf{a} : e'$ 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 [32]'s lazy semantics, which extends Launchbury [28]'s natural semantics with the *black hole* [2, 3, 32] ("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 $\mathsf{fix}_{\mathbf{G}}\ (\lambda x.\mathsf{a} : x)$ in the usual lazy evaluation is cyclic and has a similar structure to a corresponding UnCAL graph $[\![\mathbf{cycle}(\{\mathsf{a} : \&\})]\!]$. However, an extra effort is required to handle $\mathsf{fix}_{\mathbf{G}}(\lambda x.x)$ and $\mathsf{fold}(\lambda a.\lambda r.r)(e)$ for example, which are nonterminating in usual semantics.

The lazy semantics with the black hole [32] plays an important role to resolve this problem. Since $\mathsf{fix}_{\mathbf{G}}(\lambda x.x)$ 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 $\mathsf{fix}_{\mathbf{G}}(\lambda x.x)$. Still, the semantics is not sufficient for terminating evaluation of recursions such as $\mathsf{fold}(\lambda a.\lambda r.r)(e)$. To make the evaluation of recursions terminating, we adopt *memoization*. Roughly speaking, since e of $\mathsf{fold}(\lambda a.\lambda r.r)(e)$ represents a graph, the recursive call of $\mathsf{fold}(\lambda a.\lambda r.r)$ must 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

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

where $\mathsf{fold}_n e$ is replaced with $\mathsf{fold}^M e$ in which M represents a memo. The memos M are all $\emptyset$ initially, and entries are added

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

Figure 5. Reduction rules of our lazy abstract machine: we assume that $(\lambda x.e)$ is α-renamed to conform with the freshness condition.

through evaluation. Although tuples and projections are important in previous sections, we shall ignore them henceforth because they are not relevant in our technical development in the following sections; our discussions can be extended to them straightforwardly. This is the reason why $\mathsf{fold}^M e$ above does not have the subscript.

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

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:

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

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

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

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 $\langle x \mid x_1 = e_1, \ldots, x_n = e_n\rangle$ as $\mathbf{letrec}\ x_1 = e_1\ \mathbf{and}\ \ldots\ \mathbf{and}\ x_n = e_n\ \mathbf{in}\ x$. 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 $\langle x \mid \mu\rangle \to \langle x \mid \mu'\rangle$ is defined by the rules in Figure 5. The rules except the ones for $\mathsf{fold}^M e$ are just straightforward extensions of [32] with constructors.

The black hole $\bullet$ represents the apparent undefinedness yielded when the value of x is required by the evaluation of x itself [32]. A typical example is $\mathsf{fix}_{\mathbf{G}}\, (\lambda x.x)$, which will be evaluated as:

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

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

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

The reduction rules for $\mathsf{fold}^M e$ are keys in this semantics. It basically works as $\mathsf{fold}_1 e$ in Section 4.1; indeed, the first, the second and the third rules correspond to (Fold1), (Fold2) and (Fold3), respectively. The first rule also says that $\bullet$ can be seen as an exception. The second and the third rules update the memo, and the fourth rule of $\mathsf{fold}^M e$ looks up the memo if there is corresponding entry in the memo. By memoization, the problematic example in Section 1 that eliminates all the edges evaluates to $\bullet$ without major changes to the original semantics [32], as illustrated below.

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

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

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

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

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. □

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* $\langle e \mid \emptyset\rangle \to^* \langle E[\mathsf{fold}^M e\, v] \mid \mu\rangle$ *where* $M(v) = x$*. Then,* $\mu(x)$ *is a value.* □

5.3 Extracting Graphs

After an evaluation, we extract a graph from the "graph" structure of a heap. For example, for a configuration $\langle x \mid x = a : x, a = \mathsf{a}\rangle$, we obtain a graph $G = (V, E, I, O)$ with $V = \{x\}$, $E = \{(x, \mathsf{a}, x)\}$, $I = \{\& \mapsto x\}$ and $O = \emptyset$. In general, a heap may contain unevaluated expressions. If a heap contains unevaluated expressions as $\{x = (\lambda y.y)(\mathsf{a} : x)\}$, we cannot extract a graph directly from the heap. To extract a graph from a configuration $\langle x \mid \mu\rangle$, we have to ensure that $\mu(y)$ is a value for all y accessible from the root x. Formally, we say a variable x *accessible* from y in μ if (x, y) belongs to the reflexive transitive closure of the $\{(z, w) \mid w \in \mathrm{fv}(\mu(z))\}$. 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 $\langle \mathsf{elim}^{\emptyset}\, x \mid \mu\rangle$ where $\mathsf{elim}^M = \mathsf{fold}^M(\lambda a.\lambda r.\mathbf{if}\ a = a\ \mathbf{then}\ r\ \mathbf{else}\ r)$. Here, $\mathbf{if}$ is used just to evaluate a; extending the abstract machine to $\mathbf{if}$ is straightforward. The evaluation of

application $\mathsf{elim}^{\emptyset}\ x$ 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 $\langle \mathsf{elim}^{\emptyset}\ x_0 \mid x_0 = e_0 \rangle \to^* \langle \bullet \mid \mu \rangle$. 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 $\mathsf{graphify}(x_0, e_0)$ below.

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

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

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

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

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

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

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* $\langle \mathsf{elim}^{\emptyset}\ e \mid \mu \rangle$ *terminates, then* $\langle \mathsf{elim}^{M}\ e \mid \mu \rangle$ *also terminates for any* M *such that* $\mu(M(v))$ *is a value for all* $v \in \mathsf{dom}(M)$*.* □

6. Type System

In this section describes the type system that guarantees termination of $\to$ under the context $\mathsf{elim}^{\emptyset}\square$. 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

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

(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

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

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 $\mathsf{elim}^{\emptyset}\ (aInB\ bs)$. In contrast, such nonregularity does not arise for functions $insA$ itself and el in Example 1. For example, if we apply them to bs above, we have

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

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

To exclude such a problematic case, we overapproximate the number of nested applications of folds and bound it by typing. Specifically, we associate a graph type with an integer as $\mathsf{G}\langle n \rangle$, and the number is increased by fold, as $\mathsf{fold}^{M} e : \mathsf{G}\langle n \rangle \to \mathsf{G}\langle m \rangle$ where $e : \mathsf{L} \to \mathsf{G}\langle m \rangle \to \mathsf{G}\langle m \rangle$ 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, $aInB = \mathsf{fold}^{\emptyset}(\lambda z.\lambda r.z : insA\ r)$ is ill-typed; the body of fold must has the type $\mathsf{L} \to \mathsf{G}\langle m \rangle \to \mathsf{G}\langle m \rangle$ but it can only have the type $\mathsf{L} \to \mathsf{G}\langle m \rangle \to \mathsf{G}\langle m + l \rangle$ 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 $\mathsf{fix}_{\mathbf{G}}$ because $\mathsf{fix}_{\mathbf{G}}$ can also produce non-regular results,

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

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

$$\frac{}{\Gamma \vdash \mathtt{a} : \mathsf{L}}\text{T-LABEL} \qquad \frac{\Gamma \vdash e_1 : \mathsf{L} \quad \Gamma \vdash e_2 : \mathsf{G}\langle n \rangle}{\Gamma \vdash (e_1 : e_2) : \mathsf{G}\langle n \rangle}\text{T-CONS}$$

$$\frac{\{\Gamma \vdash e_i : \mathsf{G}\langle n \rangle\}_{i=1,2}}{\Gamma \vdash e_1 \cup e_2 : \mathsf{G}\langle n \rangle}\text{T-CHOICE} \qquad \frac{}{\Gamma \vdash \bullet : \tau}\text{T-BH}$$

$$\frac{\Gamma \vdash e : \mathsf{G}\langle n \rangle \to \mathsf{G}\langle n \rangle}{\Gamma \vdash \mathsf{fix}_{\mathbf{G}}\ e : \mathsf{G}\langle n \rangle}\text{T-FIX}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : \mathsf{L} \to \mathsf{G}\langle m \rangle \to \mathsf{G}\langle m \rangle \\ \{\Gamma \vdash v : \mathsf{G}\langle n \rangle \wedge \Gamma \vdash x : \mathsf{G}\langle m \rangle\}_{(v,x)\in M} \quad m > n\end{array}}{\Gamma \vdash \mathsf{fold}^{M} e : \mathsf{G}\langle n \rangle \to \mathsf{G}\langle m \rangle}\text{T-CATA}$$

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

Figure 6. Typing rules for termination.

6.2 Types

A *type* τ is defined as follows.

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

Types consist of graph types with generation ($\mathsf{G}\langle n \rangle$), function types ($\to$) and the label type ($\mathsf{L}$). It is natural to have a subtyping relation $\mathsf{G}\langle m \rangle \preceq \mathsf{G}\langle n \rangle$ 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 6 shows the typing rules. The typing rules for variables and λs are standard ones. The rules T-CONS, T-CHOICE and T-FIX says that 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 $\mathsf{fold}\,f$ 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* $\vdash g : \mathsf{DB}^X_Y \rightsquigarrow e$*, then* $\vdash e : (\mathsf{G}\langle n\rangle)^{|Y|} \to (\mathsf{G}\langle n\rangle)^{|X|}$ *for some* n. □

Notice that this translation uses general versions of $\mathsf{fix}_{\mathbf{G}}\, e$ and $\mathsf{fold}^M_k e$ extended for tuples. Here, we just note fold^M_k's general typing rule; that for $\mathsf{fix}_{\mathbf{G}}\, e$ is much straightforward.

$$\frac{\begin{array}{c}\Gamma \vdash e : \mathsf{L} \to \prod_{1\le i\le k} \mathsf{G}\langle m_i\rangle \to \prod_{1\le i\le n} \mathsf{G}\langle m_i\rangle \\ \{\Gamma \vdash v : \mathsf{G}\langle n\rangle \wedge \Gamma \vdash x : \prod_{1\le i\le k} \mathsf{G}\langle m_i\rangle\}_{(v,x)\in M} \\ k < \min\{m_i\}_{1\le i\le k}\end{array}}{\Gamma \vdash \mathsf{fold}^M_k e : \mathsf{G}\langle n\rangle \to \prod_{1\le i\le k} \mathsf{G}\langle m_i\rangle}\ \text{T-CATA}$$

Here, we write $\prod_{1\le i\le k} \tau_i$ for $\tau_1 \times \cdots \times \tau_k$. This type allows us to traverse the ith component of a tuple to produce the jth component if $m_i < m_j$, as $\mathsf{fold}^{\emptyset}_2(\lambda a.\lambda r.(\mathsf{fold}^{\emptyset} f\ (\pi_2 z), a : \pi_2 z))$, 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 [33]'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* $\vdash f : \mathsf{G}\langle n\rangle \to \mathsf{G}\langle m\rangle$*. If* $\vdash e_i : \mathsf{G}\langle n\rangle$ $(i = 1, 2)$*,* $e_1 \sim e_2$ *implies* $f\, e_1 \sim f\, e_2$. □

7. Soundness of the Type System

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

Formally, we will prove the following type soundness theorem by using logical relation [38].

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

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 [38]. 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.

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

Here, we assume that $\mathsf{dom}(\mu)$ and $\mathsf{dom}(\Gamma)$ are disjoint, 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* $\mathsf{dom}(\Delta)$*, then* $\Gamma \vdash_{\mu} e : \tau$ *holds.* □

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

The following lemma says that an evaluation of an expression of type $\mathsf{G}\langle n\rangle$ cannot "observe" graphs of type $\mathsf{G}\langle n\rangle$; recall that fold is the only language construct that can observe graphs.

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

7.2 Logical Relation

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

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

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

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

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

7.3 Lemmas for Recursive Definitions

In advance 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 $\mathsf{fix}_{\mathbf{G}}\, e$ expressions corresponds to a finite graph if e preserves finiteness.

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

Proof (Sketch). It suffices to show that $\langle \mathsf{elim}^{\emptyset}\, w \mid w = e_0\, w, \mu\rangle$ terminates. Consider $c_0 = \langle \mathsf{elim}^{\emptyset}\, w \mid w = e_0\, u, u = \bullet, \mu\rangle$ where $\mathsf{fix}_{\mathbf{G}}$ is unfolded only once. Then, we prove the statement by showing that $d_0 = \langle \mathsf{elim}^{\emptyset}\, w \mid w = e_0\, w, \mu\rangle$ terminates if c_0 does, by using Lemma 4. The termination of c_0 can be concluded from the premise of this lemma. □

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

Lemma 8 (fold)**.** *If* $\vdash_\mu \mathsf{fold}^M e :: \mathsf{G}\langle n\rangle \to \mathsf{G}\langle m\rangle$ *($m > n$) and* $\langle e \,|\, \mu\rangle \in \mathcal{R}_{\mathsf{L}\to\mathsf{G}\langle m\rangle\to\mathsf{G}\langle m\rangle}$*, then* $\langle \mathsf{fold}^M e \,|\, \mu\rangle \in \mathcal{R}_{\mathsf{G}\langle n\rangle\to\mathsf{G}\langle m\rangle}$.

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

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

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

$$\langle E[\mathsf{fold}_{(0)} e\, v] \,|\, \mu\rangle \to \langle E[\bullet] \,|\, \mu\rangle$$

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

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

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

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

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

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

Cases T-Cons**.** In this case, we have $e = e_1 : e_2$ and $\tau = \mathsf{G}\langle n\rangle$. Then, $\langle \mathsf{elim}^\emptyset\ (e_1 : e_2) \,|\, \eta\rangle$ has the reduction sequence

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

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

$$\cdots \to^* \langle \mathsf{elim}^{M''} x_2 \,|\, x_1 = \mathsf{a}, x_2 = e_2, \eta'\rangle$$

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

Case T-BH**.** By induction of τ. Note that, if we apply $\bullet$ of type $\tau_1 \to \tau_2$ 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. □

As a consequence, we have obtained Theorem 5, which says that every expression of type $\mathsf{G}\langle n\rangle$ corresponds to a finite graph, in the sense that $\to$ terminates under the observation $\mathsf{elim}^\emptyset$ □.

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

8. Related Work

We have discussed transformations of graphs up to *bisimilarity* like UnCAL [7]. Many frameworks have been proposed for transformations of graphs up to *equality/isomorphism*, from the functional programming community [9, 11–13, 17, 25, 27] and from the database community [1, 8]. 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 [9, 13, 17] 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, $\pi_1(\mathsf{fix}_{\mathbf{G}}\ (\lambda x.(\pi_2 x, \pi_1 x))$ successfully evaluates to $\bullet$ in [32] and ours. In addition, even with μ-terms, we have to exclude problematic examples such as $\mu x.\mathsf{b} : insA\ x$.

CoCaml has "recursions" with user-specified resolvers of their fixpoints [24]. 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 [33] discuss recursions similar to **srec**, for application of parallel queries on object-oriented databases [34]. Surprisingly, their idea of computation is quite similar to the bulk semantics of UnCAL [7], while general paramorphisms are not supported in their framework.

Hamana and the authors [18] 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] 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.

[5] 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.

[6] 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.

[7] 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.

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

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

[10] Agostino Dovier, Carla Piazza, and Alberto Policriti. An efficient algorithm for computing bisimulation equivalence. *Theoretical Computer Science*, 311(1):221 – 256, 2004.

[11] 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.

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

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

[14] Sebastian Fischer, Oleg Kiselyov, and Chung-chieh Shan. Purely functional lazy non-deterministic programming. In *ICFP*, pages 11–22, 2009.

[15] Andrew J. Gill, John Launchbury, and Simon L. Peyton Jones. A short cut to deforestation. In *FPCA*, pages 223–232, 1993.

[16] 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.

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

[18] 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.

[19] Soichiro Hidaka, Zhenjiang Hu, Kazuhiro Inaba, Hiroyuki Kato, Kazutaka Matsuda, and Keisuke Nakano. Bidirectionalizing graph transformations. In *ICFP*, pages 205–216. ACM, 2010.

[20] 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.

[21] Soichiro Hidaka, Kazuyuki Asada, Zhenjiang Hu, Hiroyuki Kato, and Keisuke Nakano. Structural recursion for querying ordered graphs. In *ICFP*, pages 305–318, 2013.

[22] 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.

[23] 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.

[24] Jean-Baptiste Jeannin, Dexter Kozen, and Alexandra Silva. Cocaml: Programming with coinductive types. `http://www.cs.cornell.edu/~kozen/Papers/cocaml.pdf`, 2013.

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

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

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

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

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

[30] 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.

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

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

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

[34] 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.

[35] Atsushi Ohori and Isao Sasano. Lightweight fusion by fixed point promotion. In *POPL*, pages 143–154. ACM, 2007.

[36] Yannis Papakonstantinou, Hector Garcia-Molina, and Jennifer Widom. Object exchange across heterogeneous information sources. In *ICDE*, pages 251–260. IEEE Computer Society, 1995.

[37] Morten Heine Sørensen, Robert Glück, and Neil D. Jones. A positive supercompiler. *J. Funct. Program.*, 6(6):811–838, 1996.

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

[39] 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.

[40] 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.