Initial Algebra Semantics for Cyclic Sharing Structures

Makoto Hamana

Department of Computer Science, Gunma University, Japan
hamana@cs.gunma-u.ac.jp

Abstract. Terms are a concise representation of tree structures. Since they can be naturally defined by an inductive type, they offer data structures in functional programming and mechanised reasoning with useful principles such as structural induction and structural recursion. In the case of graphs or ”tree-like” structures – trees involving cycles and sharing – however, it is not clear what kind of inductive structures exists and how we can faithfully assign a term representation of them. In this paper we propose a simple term syntax for cyclic sharing structures that admits structural induction and recursion principles. We show that the obtained syntax is directly usable in the functional language Haskell, as well as ordinary data structures such as lists and trees. To achieve this goal, we use categorical approach to initial algebra semantics in a presheaf category. That approach follows the line of Fiore, Plotkin and Turi’s models of abstract syntax with variable binding.

1 Introduction

Terms are a convenient, concise and mathematically clean representation of tree structures used in logic and theoretical computer science. In the field of traditional algorithms or graph theory, one usually uses unstructured representations for trees, such as a pair (V, E) of vertices and edges sets, adjacency lists, adjacency matrices, pointer structures, etc, which are more complex and unreadable than terms. We know that term representation provides a well-structured, compact and more readable notation.

However, consider the case of “tree-like” structures such as that depicted in Fig. 1. This kind of structures – graphs, but almost trees involving (a few) exceptional edges – quite often appears in logic and computer science. Examples include internal representations of expressions in implementations of functional languages that share common subexpressions for efficiency, control flow graphs of imperative programs used in static analysis and compiler optimizations [CFR+91], data models of XML such as trees with pointers [CGZ05], proof trees admitting cycles for cyclic proofs [Bro05], and term graphs in graph rewriting [BvEG+87, AK96].

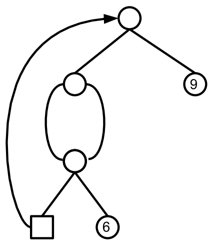

Fig. 1.

Suppose we want to treat such structures in a pure functional programming language such as Haskell, Clean, or a proof assistant such as Coq, Agda [Nor07]. In such a case, we would have to abandon the use of naive term representation, and would instead be compelled to use an unstructured representation such as (V, E), adjacency lists, etc. Furthermore, a serious problem is that we would have to abandon structural recursion/induction to decompose them because they look "tree-like" but are in fact graphs, so there is no obvious inductive structure in them. This means that in functional programming, we cannot use pattern matching to treat tree-like structures, which greatly decreases their convenience. This lack of structural induction means failure of being an inductive type. But, are there really no inductive structures in tree-like structures? As might be readily apparent, tree-like structures are almost trees and merely contain finite pieces of information. The only difference is the presence of "cycles" and "sharing".

In this paper, we give an initial algebra characterisation of cyclic sharing structures in the framework of categorical universal algebra. The aim of this paper is to derive the following practical goals *from the initial algebra characterisation*.

[I] To develop a simple term syntax for cyclic sharing structures that admits structural induction and structural recursion principles.
[II] To make the obtained syntax directly usable in the current functional languages and proof assistants, as well as ordinary data structures, such as lists and trees.

The goal [I] also intends that the term syntax *exactly* characterises cyclic sharing structures (i.e. no junk terms exist) to make structural induction possible. The goal [II] intends that the obtained syntax should be *lightweight as possible*, which means that e.g. well-formedness and equality tests on terms for cyclic sharing structures should be fast and easy, as are ordinary data structures such as lists and trees. We do not want many axioms to characterise the intended structures, because in programming situation, to check the validity of axioms every time is expensive and makes everything complicated. Therefore, ideally, formulating structures *without axioms* is best. The goal [II] is rephrased more specifically as:

[II'] To give an inductive type that represents cyclic sharing structures uniquely. We therefore rely on that a type checker automatically ensures the well-formedness of cyclic sharing structures.

We choose a functional programming language Haskell to concretely show it in this paper. To archive these goals, we use the category theoretic formulation of initial algebra semantics.

Recently, varying the base category other than **Set**, initial algebra semantics for functor-algebras has proved to be useful framework to characterise various mathematical/computational structures in a uniform setting. We list several: S-sorted abstract syntax is characterised as initial algebra in $\mathbf{Set}^S$ [Rob02], second-order abstract syntax as initial algebra in $\mathbf{Set}^{\mathbb{F}}$ [FPT99, Ham04, Fio08] (where $\mathbb{F}$ is the category of finite sets), explicit substitutions as initial algebras in the category $[\mathbf{Set},\mathbf{Set}]_f$ of finitary functors [GUH06], recursive path ordering for term rewriting systems as algebras in the category **LO** of linear orders [Has02], nested datatypes [GJ07] and generalised algebraic datatypes (GADTs) [JG08] in functional programming as initial algebras in $[C,C]$ and $[|C|,C]$ respectively, where C is a ω-cocomplete category.

This paper adds a further example to the above list. We characterise cyclic sharing structures as an initial algebra in the category $(\mathbf{Set}^{\mathbb{T}^*})^{\mathbb{T}}$, where $\mathbb{T}$ is the set of all "shapes" of trees and $\mathbb{T}^*$ is the set of all tree shape contexts. We derive structural induction and recursion principles from it. An important point is that we merely use *algebra of functor* to formulate cyclic sharing structures, i.e. not (models of) equational specifications or (Σ, E)-algebras. This achieves the requirement of "without axioms" and is the key to formulate them by an inductive type.

Basic idea. It is known in the field of graph algorithms [Tar72] that, by traversing a rooted directed graph in a depth-first search manner, we obtain a *depth-first search tree*, which consists of a spanning tree (whose edges are called *tree edges*) of the graph and *forward edges* (which connect ancestors in the tree to descendants), *back edges* (the reverse), and *cross edges* (which connect nodes across the tree from right to left).

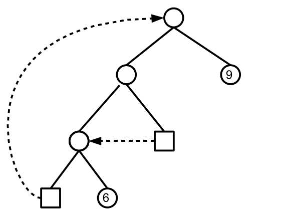

Fig. 2. Depth-first search tree

Forward edges can be decomposed into tree and cross edges by placing indirect nodes. For example, the graph in Fig. 1 becomes a depth-first search tree in Fig. 2 where solid lines are tree edges and dashed lines are back and cross edges. This is the target structure we will model in this paper. That is, tree edges are the basis of an *inductive structure*, back edges are to form *cycles*, and cross edges are to form *sharing*. Consequently, our task is to seek how to characterise pointers making back edges and cross edges in inductive constructions.

Formulation. The crucial idea to formulate pointers in inductive constructions is to use *binders* as *pointers* in abstract syntax. Trees are formulated as terms. Hence, a remaining problem is how to exactly capture binders in terms. Fiore, Plotkin and Turi [FPT99] has characterised abstract syntax with variable binding by an initial algebra in the category $\mathbf{Set}^{\mathbb{F}}$. For example, abstract syntax of λ-terms is modeled as a functor

$$\Lambda : \mathbb{F} \longrightarrow \mathbf{Set}$$

equipped with three constructors for λ-terms as an algebra structure on Λ. Each set $\Lambda(X)$ gives the set of all λ-terms which may contain free variables taken from a set X in $\mathbb{F}$. This formulation models a structure (here, abstract syntax trees) indexed by suitable invariant (here, free variables considered as contexts), which is essential information to capture the intended structure (abstract syntax with variable binding).

However, this approach using algebras in $\mathbf{Set}^{\mathbb{F}}$ is insufficient to represent "cross edges" in tree-like graphs. Ariola and Klop has analysed that there are two kinds of sharing in this kind of tree-like graphs [AK96]: (i) vertical sharing (i.e. back edges in depth-first search trees) and (ii) horizontal sharing (i.e. cross edges). In principle, binders capture "vertical" contexts only, but to represent cross edges exactly, we must capture "horizontal" context information, which cannot be handled by the index category $\mathbb{F}$.

To solve this problem, in this paper we take a richer index category that is enough to model cross edges. We introduce the notion of *shape trees* and contexts consisting of them, which represents other parts of a tree viewing from a pointer node. We use shape trees $\mathbb{T}$ as "types" of syntax and $\mathbb{T}^*$ as "context". We use Fiore's initial algebra semantics for typed abstract syntax with variable binding [Fio02] by algebras in the presheaf

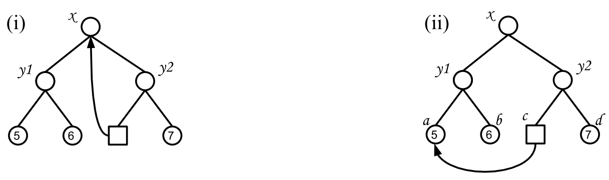

Fig. 3. Trees involving cycle and sharing

category $(\mathbf{Set}^{\mathbb{T}^*})^{\mathbb{T}}$. Hence cyclic sharing trees are modeled as a $\mathbb{T}$ and $\mathbb{T}^*$-indexed set

$$T : \mathbb{T} \longrightarrow (\mathbb{T}^* \longrightarrow \mathbf{Set})$$

equipped with constructors of cyclic sharing trees as an algebra structure.

Organisation. We first give types and abstract syntax for cyclic sharing binary trees in Section 2. We then characterise cyclic sharing binary trees as an initial algebra in Section 3. Section 4 gives a way of implementing cyclic sharing structures by an inductive type in Haskell. Section 5 generalises our treatment to arbitrary signature for cyclic sharing structures. Section 6 relates our representation and equational term graphs in the initial algebra framework by giving a homomorphic translation. In Section 7, we discuss connections to other approaches to cyclic sharing structures.

2 Abstract Syntax for Cyclic Sharing Structures

Cyclic structures by μ-terms. The syntax of fixpoint expressions by μ-notation (μ-terms) is widely used in computer science and logic. Its theory has been thoroughly investigated, e.g., in [AK96]. The language of μ-terms suffices to express all cyclic structures.

For example, a cyclic binary tree shown in Fig. 3 (i) is representable by the term

$$\mu x.\mathsf{bin}(\mu y_1.\mathsf{bin}(\mathsf{lf}(5), \mathsf{lf}(6)),\ \mu y_2.\mathsf{bin}(\ x,\ \mathsf{lf}(7)\,))\tag{1}$$

where bin and lf denote binary and leaf node, respectively. The point is that the variable x refers to the root labeled by a μ-binder, hence a cycle is represented. To uniquely formulate cyclic structures, here we introduce the following assumption: we attach μ-binders in front of bin only, and put exactly one μ-binder for each occurrence of bin as for (1). This is seen as uniform addressing of bin-node, i.e., x, y_1, y_2 are seen as as labels or "addresses" of bin-nodes. We also assume no axiom to equate μ-terms. That is, we do not identify a μ-term with its unfolding, since they are different (shapes of) graphs. In summary, μ-terms represent cyclic structures. This is the underlying idea of representation of cyclic data in a functional programming language Haskell [GHUV06].

How to represent sharing. Next, we incorporate sharing. The presence of sharing makes the situation more difficult. Consider the tree (ii) in Fig. 3 involving sharing via a cross edge. As similar to the case of cycles, this might be written as a μ-term

$$\mu x.\mathsf{bin}(\mu y_1.\mathsf{bin}(\mathsf{lf}(5), \mathsf{lf}(6)),\ \mu y_2.\mathsf{bin}(\boxed{}, \mathsf{lf}(7))).$$

But can we fill the blank to refer the node a (in Fig. 3 (ii)) from the node c "horizontally" by using the mechanism of binders? Actually, μ-binders are insufficient for this purpose. Hence, we introduce a new notation "$\swarrow p \uparrow x$" to refer to a node horizontally. This notation means going up to a node x labelled by a μ-binder and going down to a position p in the subtree rooted by the node x. In the above example, the blank is filled by $\swarrow 11 \!\uparrow\! x$, which means going back to the node x, then going down through the left child twice (using the position 11). See also Example 1. We firstly focus on the formulation of binary trees involving cycles and sharing. Binary trees are the minimal case that can involve the notion of sharing in structures. To ensure correct sharing, we introduce the notion of shape trees.

Shape trees. We call our target data structures *cyclic sharing trees* and its syntactic representation *cyclic sharing terms*. Cyclic sharing trees are binary trees generated by three kinds of nodes, that is, pointer, leaf, and binary node, and satisfying a certain well-formedness condition.

We use skeletons of cyclic sharing trees, called *shape trees*. Shape trees are binary trees forgetting values in pointer and leaf nodes from cyclic sharing trees. The set $\mathbb{T}$ of all shape trees is defined by

$$\mathbb{T} \ni \tau ::= \textsc{e} \mid \textsc{p} \mid \textsc{l} \mid \textsc{b}(\tau_1, \tau_2)$$

where $\textsc{e}$ is the void shape, $\textsc{p}$ is the shape of pointer node, $\textsc{l}$ is the shape of leaf node, and $\textsc{b}(\tau_1, \tau_2)$ is the shape of a binary node. We typically use Greek letters σ, τ to denote shape trees.

We define referable positions in a shape tree. A *position* is a finite sequence of $\{1, 2\}$. The root position is denoted by the empty sequence ϵ and the concatenation of positions is denoted by pq or $p.q$. The set $\mathcal{P}os(\tau)$ of positions in a shape tree τ is defined by $\mathcal{P}os(\textsc{e}) = \mathcal{P}os(\textsc{p}) = \emptyset$, $\mathcal{P}os(\textsc{l}) = \{\epsilon\}$, $\mathcal{P}os(\textsc{b}(\sigma, \tau)) = \{\epsilon\} \cup \{1p \mid p \in \mathcal{P}os(\sigma)\} \cup \{2p \mid p \in \mathcal{P}os(\tau)\}$. An important point is that the void $\textsc{e}$ and the pointer $\textsc{p}$ nodes are not referable by other nodes, hence their positions are defined to be empties.

Syntax and types. Shape trees are used as types in a typing judgment. As usual, a typing context Γ is a sequence of (variable, shape tree)-pairs.

Typing rules (named binder version)

$$(\text{Pointer})\frac{p \in \mathcal{P}os(\sigma)}{\Gamma, x : \sigma, \Gamma' \vdash \swarrow p \!\uparrow\! x : \textsc{p}} \qquad (\text{Leaf})\frac{k \in \mathbb{Z}}{\Gamma \vdash \mathsf{lf}(k) : \textsc{l}}$$

$$(\text{Node})\frac{x : \textsc{b}(\textsc{e}, \textsc{e}), \Gamma \vdash s : \sigma \quad x : \textsc{b}(\sigma, \textsc{e}), \Gamma \vdash t : \tau}{\Gamma \vdash \mu x.\mathsf{bin}(s, t) : \textsc{b}(\sigma, \tau)}$$

In these typing rules, a shape tree type is assigned to the corresponding tree node. That is, a binary node is of type $\textsc{b}(\sigma, \tau)$ of binary node shape, a pointer node is of type $\textsc{p}$ of pointer node shape, and a leaf node is of type $\textsc{l}$ of leaf node shape.

A type declaration $x : \sigma$ in a typing context (roughly) means that σ is the shape of subtree (say, t) headed by a binder μx (see Example 1). Hence, in (Pointer) rule, taking a position $p \in \mathcal{P}os(\sigma)$, we safely refer to a position in the tree t. The notation $\swarrow p {\uparrow} x$ is designed to realise a right-to-left cross edge. Note also that a path obtained by $\swarrow p {\uparrow} x$ is the shortest path from the pointer node to the node referred by $\swarrow p {\uparrow} x$. When $p = \epsilon$, we abbreviate $\swarrow \epsilon {\uparrow} x$ as ${\uparrow} x$. This ${\uparrow} x$ exactly expresses a back edge. In (Node) rule, the shape trees $\textsc{b}(\textsc{e}, \textsc{e})$ and $\textsc{b}(\sigma, \textsc{e})$ mask nodes that are reachable via left-to-right references (i.e. not our requirement) or redundant references (e.g. going up to a node x then going back down through the same path) by the void shape $\textsc{e}$.

Example 1. The binary tree involving sharing in Fig. 3 (ii) is represented by a well-typed term $\mu x.\mathsf{bin}(\mu y_1.\mathsf{bin}(\mathsf{lf}(5), \mathsf{lf}(6)),\ \mu y_2.\mathsf{bin}(\swarrow 11 \uparrow x, \mathsf{lf}(7)))$ Typing derivation is as follows.

$$\dfrac{\dfrac{y_1{:}\alpha, x{:}\alpha \vdash \mathsf{lf}(5) : \textsc{l} \quad y_1{:}\textsc{b}(\textsc{l},\textsc{e}), x{:}\alpha \vdash \mathsf{lf}(6) : \textsc{l}}{x{:}\alpha \vdash \mu y_1.\mathsf{bin}(\mathsf{lf}(5), \mathsf{lf}(6)) : \textsc{b}(\textsc{l},\textsc{l})} \quad \dfrac{\dfrac{11 \in \mathcal{P}os(\beta)}{y_2{:}\alpha, x{:}\beta \vdash \swarrow 11{\uparrow}x : \textsc{p}} \quad y_2{:}\textsc{b}(\textsc{p},\textsc{e}), x{:}\beta \vdash \mathsf{lf}(7) : \textsc{l}}{x{:}\beta \vdash \mu y_2.\mathsf{bin}(\swarrow 11{\uparrow}x, \mathsf{lf}(7)) : \textsc{b}(\textsc{p},\textsc{l})}}{\vdash \mu x.\mathsf{bin}(\mu y_1.\mathsf{bin}(\mathsf{lf}(5, \mathsf{lf}(6)), \mu y_2.\mathsf{bin}(\swarrow 11{\uparrow}x, \mathsf{lf}(7))) : \textsc{b}(\textsc{b}(\textsc{l},\textsc{l}), \textsc{b}(\textsc{p},\textsc{l}))}$$

where $\alpha = \textsc{b}(\textsc{e}, \textsc{e})$, $\beta = \textsc{b}(\textsc{b}(\textsc{l}, \textsc{l}), \textsc{e})$.

Instead of named variables for binders, a de Bruijn notation is also possible. The construction rules are reformulated as follows. Now a typing context Γ is simply a sequence of shape trees $\tau_1, \ldots, \tau_n$. Let $|\Gamma|$ denote its length. A judgment $\Gamma \vdash t : \tau$ denotes a well-formed term t of shape τ containing free variables (de Bruijn indices) from 1 to $|\Gamma|$. The intended meaning is that the length $|\Gamma|$ denotes how many maximally we can go up from the current node t, and each shape tree τ_i in Γ denotes the shape of the subtree at i-th upped node from t. Therefore, when t is a pointer, a context specifies the set of all positions that a pointer node can refer to.

As known from λ-calculus, using de Bruijn notation, binders become nameless, so we can safely omit "x" from μx. Since the typing rules are designed to attach exactly one μ-binder for each bin, even "μ" can be omitted. Hence we have a simplified construction rules of terms.

Typing rules (de Bruijn version)

$$(\text{dbPointer})\dfrac{|\Gamma| = i - 1 \quad p \in \mathcal{P}os(\sigma)}{\Gamma, \sigma, \Gamma' \vdash \swarrow p{\uparrow}i : \textsc{p}} \qquad (\text{dbLeaf})\dfrac{k \in \mathbb{Z}}{\Gamma \vdash \mathsf{lf}(k) : \textsc{l}}$$

$$(\text{dbNode})\dfrac{\textsc{b}(\textsc{e},\textsc{e}), \Gamma \vdash s : \sigma \quad \textsc{b}(\sigma,\textsc{e}), \Gamma \vdash t : \tau}{\Gamma \vdash \mathsf{bin}(s, t) : \textsc{b}(\sigma, \tau)}$$

In (dbPointer) rule, the condition $|\Gamma| = i - 1$ says that the shape tree σ appears at i-th position of the typing context in the lower judgment. Since for a given graph its depth-first search tree is unique, the following is immediate.

Theorem 1. *Given rooted graph which is connected, directed and edge-ordered, the term representation in de Bruijn is unique.*

Remark 1. This uniqueness of term representation has practical importance. For instance, for the graph in the tree (ii) in Fig. 3, there is only one way to represent it in this term syntax, i.e., $\mathsf{bin}(\mathsf{bin}(\mathsf{lf}(5, \mathsf{lf}(6)), \mathsf{bin}(\swarrow 11{\uparrow}1, \mathsf{lf}(7)))$ in de Bruijn. Hence, we do not need any complex equality on graphs (other than the syntactic equality) to check whether a given data is the required one. This is in contrast to other approaches. If we represent a graph as a term graph with labels [BvEG$^+$87], an equational term graph [AK96], or a **letrec**-term [Has97], there are several syntactic representations for a single graph, hence some normalisation is required when e.g., defining a function on graphs. Generally speaking, our terms are seen as "de Bruijn notation" of term graphs with labels [BvEG$^+$87].

3 Initial Algebra Semantics

3.1 Construction

In this section, we show that cyclic sharing terms form an initial algebra and derive structural recursion and induction from it. We use Fiore's approach to algebras for typed abstract syntax with binding [Fio02] in the presheaf category $(\mathbf{Set}^{\mathbb{F}\downarrow U})^U$ where U is the set of all types. Now, we take the set $\mathbb{T}$ of all shape trees for U, and the set $\mathbb{N}$ of natural numbers for variables (i.e. pointers), instead of the category $\mathbb{F}$ of finite sets and all functions (used for renaming variables), because we do not need renaming of pointers.

Algebras. We define the discrete category $\mathbb{T}^*$ by taking contexts $\Gamma = \langle \tau_1, \ldots, \tau_n \rangle$ as objects (which is equivalent to $\mathbb{N} \downarrow \mathbb{T}$). We consider algebras in $(\mathbf{Set}^{\mathbb{T}^*})^{\mathbb{T}}$. Two preliminary definitions. We define the presheaf $\mathrm{PO} \in \mathbf{Set}^{\mathbb{T}^*}$ for pointers by

$$\mathrm{PO}(\langle \tau_1, \ldots, \tau_n \rangle) = \{\swarrow p{\uparrow}i \mid 1 \le i \le n,\ p \in \mathcal{P}os(\tau_i)\}.$$

For each $\tau \in \mathbb{T}$, we define the functor $\delta_\tau : \mathbf{Set}^{\mathbb{T}^*} \longrightarrow \mathbf{Set}^{\mathbb{T}^*}$ for context extension by $\delta_\tau A = A(\langle \tau, - \rangle)$.

We define the *signature functor* $\Sigma : (\mathbf{Set}^{\mathbb{T}^*})^{\mathbb{T}} \longrightarrow (\mathbf{Set}^{\mathbb{T}^*})^{\mathbb{T}}$ *for cyclic sharing binary trees*, which takes $A \in (\mathbf{Set}^{\mathbb{T}^*})^{\mathbb{T}}$ and a type in $\mathbb{T}$, and gives a presheaf in $\mathbf{Set}^{\mathbb{T}^*}$, as follows:

$$(\Sigma A)_{\mathrm{E}} = 0 \qquad (\Sigma A)_{\mathrm{P}} = \mathrm{PO} \qquad (\Sigma A)_{\mathrm{L}} = K_{\mathbb{Z}} \qquad (\Sigma A)_{\mathrm{B}(\sigma,\tau)} = \delta_{\mathrm{B}(\mathrm{E},\mathrm{E})} A_\sigma \times \delta_{\mathrm{B}(\sigma,\mathrm{E})} A_\tau$$

where $K_{\mathbb{Z}}$ is the constant functor to $\mathbb{Z}$, and 0 is the empty set functor. A Σ-*algebra* A is a pair (A, α) consisting of a presheaf $A \in (\mathbf{Set}^{\mathbb{T}^*})^{\mathbb{T}}$ for a carrier and a natural transformation $\alpha : \Sigma A \to A$ for an algebra structure. By definition of Σ, to give an algebra structure is to give the following morphisms of $\mathbf{Set}^{\mathbb{T}^*}$:

$$\mathsf{ptr}^A : \mathrm{PO} \to A_{\mathrm{P}} \qquad \mathsf{lf}^A : K_{\mathbb{Z}} \to A_{\mathrm{L}} \qquad \mathsf{bin}^{\sigma,\tau A} : \delta_{\mathrm{B}(\mathrm{E},\mathrm{E})} A_\sigma \times \delta_{\mathrm{B}(\sigma,\mathrm{E})} A_\tau \to A_{\mathrm{B}(\sigma,\tau)}.$$

A *homomorphism* of Σ-algebras is a map $\phi : (A, \alpha) \to (B, \beta)$ such that $\phi \circ \alpha = \beta \circ \Sigma\phi$.

Initial Algebra. Let T be the presheaf of all derivable cyclic sharing terms defined by

$$T_\tau(\Gamma) = \{t \mid \Gamma \vdash t : \tau\}.$$

Theorem 2. *For the signature functor Σ for cyclic sharing binary trees, T forms an initial Σ-algebra.*

Proof. Since δ_τ preserves ω-colimits, so does Σ. An initial Σ-algebra is constructed by the colimit of the ω-chain $0 \to \Sigma 0 \to \Sigma^2 0 \to \cdots$ [SP82]. These construction steps correspond to derivations of terms by typing rules, hence their union T is the colimit. The algebra structure $\mathsf{in} : \Sigma T \to T$ of the initial algebra is obtained by one-step inference of the typing rules, i.e., given by the following operations

$$\mathsf{ptr}^T(\Gamma) : \mathrm{PO}(\Gamma) \to T_{\mathrm{P}}(\Gamma) \qquad \mathsf{lf}^T(\Gamma) : \mathbb{Z} \to T_{\mathrm{L}}(\Gamma)$$
$$\swarrow p{\uparrow}i \mapsto \swarrow p{\uparrow}i \qquad\qquad k \mapsto \mathsf{lf}(k)$$
$$\mathsf{bin}^T(\Gamma) : T_\sigma(\mathrm{B}(\mathrm{E},\mathrm{E}),\Gamma) \times T_\tau(\mathrm{B}(\sigma,\mathrm{E}),\Gamma) \to T_{\mathrm{B}(\sigma,\tau)}(\Gamma); \qquad s,\, t \mapsto \mathsf{bin}(s,t). \qquad \square$$

This development of initial algebra characterisation follows the line of [FPT99, Fio02, MS03]. Hence we can further develop full theory of algebraic models of abstract syntax for cyclic sharing structures along this line. It will provide second-order typed abstract syntax with object/meta-level variables and substitutions via a substitution monoidal structure and a free Σ-monoid [Ham04, Fio08] in $(\mathbf{Set}^{\mathbb{T}^*})^{\mathbb{T}}$ (by incorporating suitable arrows into $\mathbb{T}^*$). Object/meta-substitutions on cyclic sharing structures will provide ways to construct cyclic sharing structures from smaller structures in a sensible manner. But this is not the main purpose of this paper, hence details will be pursued elsewhere.

3.2 Structural Recursion Principle

One of the important benefit of initial algebra characterisation is that the unique homomorphism from the initial to another algebra is a mapping defined by structural recursion.

Theorem 3. *The unique homomorphism ϕ from the initial Σ-algebra T to a Σ-algebra A is described by*

$$\phi_{\mathrm{P}}(\Gamma)(\swarrow p{\uparrow}i) = \mathsf{ptr}^A(\Gamma)(\swarrow p{\uparrow}i)$$
$$\phi_{\mathrm{L}}(\Gamma)(\mathsf{lf}(k)) = \mathsf{lf}^A(\Gamma)(k)$$
$$\phi_{\mathrm{B}(\sigma,\tau)}(\Gamma)(\mathsf{bin}(s,t)) = \mathsf{bin}^A(\Gamma)(\phi_\sigma(\mathrm{B}(\mathrm{E},\mathrm{E}),\Gamma)(s),\ \phi_\tau(\mathrm{B}(\sigma,\mathrm{E}),\Gamma)(t))$$

Proof. Since the unique homomorphism $\phi : T \longrightarrow A$ is a morphism of $(\mathbf{Set}^{\mathbb{T}^*})^{\mathbb{T}}$. $\square$

Example 2. We define (1) the function height that computes the height of a cyclic sharing tree $t \in T_\tau(\Gamma)$ (2) the function leaves that collects all leaf values in t by structural recursion (where max is the maximal function):

$$\mathsf{height}_{\mathrm{P}}(\Gamma)(\swarrow p{\uparrow}i) = 1 \qquad \mathsf{height}_{\mathrm{L}}(\Gamma)(\mathsf{lf}(k)) = 1$$
$$\mathsf{height}_{\mathrm{B}(\sigma,\tau)}(\Gamma)(\mathsf{bin}(s,t)) = \mathsf{max}(\mathsf{height}_\sigma(\mathrm{B}(\mathrm{E},\mathrm{E}),\Gamma)(s), \mathsf{height}_\tau(\mathrm{B}(\sigma,\mathrm{E}),\Gamma)(t))$$
$$\mathsf{leaves}_{\mathrm{P}}(\Gamma)(\swarrow p{\uparrow}i) = \emptyset \qquad \mathsf{leaves}_{\mathrm{L}}(\Gamma)(\mathsf{lf}(k)) = \{k\}$$
$$\mathsf{leaves}_{\mathrm{B}(\sigma,\tau)}(\Gamma)(\mathsf{bin}(s,t)) = \mathsf{leaves}_\sigma(\mathrm{B}(\mathrm{E},\mathrm{E}),\Gamma)(s) \cup \mathsf{leaves}_\tau(\mathrm{B}(\sigma,\mathrm{E}),\Gamma)(t)$$

This is because leaves is the unique homomorphism from T to a Σ-algebra $K_{\mathbb{Z}}$ whose operations are $\mathsf{ptr}^{K_{\mathbb{N}}}(\Gamma)(\swarrow p{\uparrow}i) = \varnothing$, $\mathsf{lf}^{K_{\mathbb{N}}}(\Gamma)(k) = \{k\}$, $\mathsf{bin}^{K_{\mathbb{N}}}(\Gamma)(x, y) = x \cup y$. Similarly for height. Notice that the height is not so directly defined in ordinary graph representations.

From algorithmic point of view, this structural recursion principle provides depth-first search traversal of a rooted graph. Hence, graph algorithms based on depth-first search are directly programmable by this structural recursion. In the author's home page (`http://www.cs.gunma-u.ac.jp/~hamana/`), several other simple graph algorithms have been programmed by structural recursion.

3.3 Structural Induction Principle

Another important benefit of initial algebra characterisation is the tight connection to structural induction principle. In order to derive it, following [HJ98], we use the category $\mathrm{Sub}((\mathbf{Set}^{\mathbb{T}^*})^{\mathbb{T}})$ for predicates on $(\mathbf{Set}^{\mathbb{T}^*})^{\mathbb{T}}$ defined by

- objects: sub-presheaves $(Q \hookrightarrow V)$, i.e., inclusions between $Q, V \in (\mathbf{Set}^{\mathbb{T}^*})^{\mathbb{T}}$,
- arrows: $u : (Q \hookrightarrow V) \to (P \hookrightarrow U)$ are natural transformations $u : V \to U$ between underlying presheaves satisfying $a \in Q_\tau(\Gamma)$ implies $u(a) \in P_\tau(\Gamma)$ for all $\tau \in \mathbb{T}, \Gamma \in \mathbb{T}^*$.

A sub-presheaf $(P \hookrightarrow T)$ is seen as a predicate P on cyclic sharing terms T, which is indexed by types and contexts. So, we say "$P^\Gamma_\tau(t)$ holds" when $t \in P_\tau(\Gamma)$ for $t \in T_\tau(\Gamma)$. We consider Σ-algebras in $\mathrm{Sub}((\mathbf{Set}^{\mathbb{T}^*})^{\mathbb{T}})$. The signature functor $\Sigma : (\mathbf{Set}^{\mathbb{T}^*})^{\mathbb{T}} \to (\mathbf{Set}^{\mathbb{T}^*})^{\mathbb{T}}$ is lifted to $\Sigma_{\mathrm{pred}} : \mathrm{Sub}((\mathbf{Set}^{\mathbb{T}^*})^{\mathbb{T}}) \to \mathrm{Sub}((\mathbf{Set}^{\mathbb{T}^*})^{\mathbb{T}})$, called the logical predicate lifting in [HJ98], by the same way to Σ:

$$(\Sigma_{\mathrm{pred}}(Q \hookrightarrow V))_{\mathrm{P}} = (\mathrm{PO} \hookrightarrow \mathrm{PO})$$

$$(\Sigma_{\mathrm{pred}}(Q \hookrightarrow V))_{\mathrm{E}} = (0 \hookrightarrow 0) \qquad (\Sigma_{\mathrm{pred}}(Q \hookrightarrow V))_{\mathrm{L}} = (K_{\mathbb{Z}} \hookrightarrow K_{\mathbb{Z}})$$

$$(\Sigma_{\mathrm{pred}}(Q \hookrightarrow V))_{\mathrm{B}(\sigma,\tau)} = \delta_{\mathrm{B}(\mathrm{E},\mathrm{E})}(Q \hookrightarrow V)_\sigma \times \delta_{\mathrm{B}(\sigma,\mathrm{E})}(Q \hookrightarrow V)_\tau$$

where the context extension is also lifted to $\delta_\tau : \mathrm{Sub}(\mathbf{Set}^{\mathbb{T}^*}) \to \mathrm{Sub}(\mathbf{Set}^{\mathbb{T}^*})$ defined by $\delta_\tau(A \hookrightarrow B) = (A\langle\tau, -\rangle \hookrightarrow B\langle\tau, -\rangle)$. Now, a Σ_{pred}-algebra structure $\alpha : \Sigma_{\mathrm{pred}}(P \hookrightarrow T) \to (P \hookrightarrow T)$ can be read as the induction hypothesis, e.g.

$$\begin{aligned}\mathsf{bin}^{\sigma,\tau\,P}(\Gamma) : (P_\sigma(\mathrm{B}(\mathrm{E},\mathrm{E}),\Gamma) \hookrightarrow T_\sigma(\mathrm{B}(\mathrm{E},\mathrm{E}),\Gamma)) \times (P_\tau(\mathrm{B}(\sigma,\mathrm{E}),\Gamma) \hookrightarrow T_\tau(\mathrm{B}(\sigma,\mathrm{E}),\Gamma))\\ \to (P_{\mathrm{B}(\sigma,\tau)}(\Gamma) \hookrightarrow T_{\mathrm{B}(\sigma,\tau)}(\Gamma)); \qquad s, t \mapsto \mathsf{bin}(s, t)\end{aligned}$$

means that "if $P^{\Gamma,\mathrm{B}(\mathrm{E},\mathrm{E})}_\sigma(s)$ & $P^{\Gamma,\mathrm{B}(\sigma,\mathrm{E})}_\tau(t)$ holds, then $P^\Gamma_{\mathrm{B}(\sigma,\tau)}(\mathsf{bin}(s,t))$ holds."

By [HJ98], $(T \hookrightarrow T)$ is an initial Σ_{pred}-algebra. The unique homomorphism $\phi : (T \hookrightarrow T) \longrightarrow (P \hookrightarrow T)$ means that P holds for all cyclic sharing terms in T. Hence

Theorem 4. *Let P be a predicate on T. To prove that $P^\Gamma_\tau(t)$ holds for all $t \in T_\tau(\Gamma)$, it suffices to show*

(i) $P^\Gamma_{\mathrm{P}}(\swarrow p{\uparrow}i)$ *holds for all* $\swarrow p{\uparrow}i \in \mathrm{PO}(\Gamma)$,
(ii) $P^\Gamma_{\mathrm{L}}(\mathsf{lf}(k))$ *holds for all* $k \in \mathbb{Z}$,
(iii) if $P^{\mathrm{B}(\mathrm{E},\mathrm{E}),\Gamma}_\sigma(s)$ & $P^{\mathrm{B}(\sigma,\mathrm{E}),\Gamma}_\tau(t)$ *holds, then* $P^\Gamma_{\mathrm{B}(\sigma,\tau)}(\mathsf{bin}(s,t))$ *holds.*

This structural induction principle can be used to prove properties of functions on cyclic sharing terms defined by structural recursion.

4 Inductive Type for Cyclic Sharing Structures

In this section, we realise our aim [II] to give an inductive type for cyclic sharing structures. We choose the functional language Haskell, because (1) we show that our characterisation of cyclic sharing is available in today's programming language technology, and (2) Haskell's type system is powerful enough to faithfully implement our initial algebra characterisation. The resulting definition might be close to implementations in proof assistants using dependent types such as Coq and Agda.

Since the set $T_\tau(\Gamma)$ of cyclic sharing terms depends on a shape tree and context, it should be implemented by a dependent type. We have seen in the proof of Thm. 2, constructors of cyclic sharing terms are $\mathbb{T}$ and $\mathbb{T}^*$-indexed functions. Inductive types defined by indexed constructors have been known as *inductive families* in dependent type theories [Dyb94]. Recently, Glasgow Haskell Compiler incorporates this feature as *GADTs* (generalised algebraic data types) [PVWW06]. With another feature called type classes, we can realise lightweight dependently-typed programming in Haskell [McB02].

We will implement $T_\tau(\Gamma)$ as a GADT "`T n t`" that depends on a context `n` (for Γ) and a shape tree `t` (for τ). Since in Haskell, a type can only depend on types (not values), we firstly define type-level shape trees by using a type class.

```
data E; data P; data L = StopLf; data B a b = DnL a | DnR b | StopB
class Shape t
instance Shape E; instance Shape P; instance Shape L
instance (Shape s, Shape t) => Shape (B s t)
```

These define constructors of shape trees as types `E`,`P`,`L` and a type constructor `B`, then group them by the type class `Shape`. Values of a shape tree τ (e.g. `B (B L L) L` in Haskell) are $\mathcal{P}os(\tau)$ (e.g. `DnL (DnR StopLf)`), i.e. "referable positions" in τ.

Similarly, a context $\langle\tau_1, \ldots, \tau_n\rangle$ is coded as a type-level sequence `TyCtx` τ_n (`TyCtx` $\tau_{n-1} \cdots$ (`TyCtx` τ_1 `TyEmp`)), and the type constructors are grouped by the type class `Ctx`. Values of a context type are "pointers" (e.g. `(Up UpStop)` meaning $\uparrow 2$).

```
data TyEmp; data TyCtx t n = Up n | UpStop | UpGD t
class Ctx n
instance Ctx TyEmp; instance (Shape t, Ctx n) => Ctx (TyCtx t n)
```

Finally, we define $T_\tau(\Gamma)$ by a GADT "T" that takes a context and a shape tree as two arguments of types.

```
data T :: * -> * -> * where
  Ptr :: Ctx n => n -> T n P
  Lf  :: Ctx n => T n L
  Bin :: (Ctx n, Shape s, Shape t) =>
         T (TyCtx (B E E) n) s -> T (TyCtx (B s E) n) t -> T n (B s t)
```

This defines three constructors of cyclic sharing terms faithfully (the part "`Ctx n` =>" is a quantification meaning that "for every type `n` which is an instance of the type class `Ctx`").

For example, the term in Example 1 is certainly a well-typed term and its type is inferred in Haskell:

```
Bin (Bin (Lf 5) (Lf 6))
    (Bin (Ptr (Up (UpGD (DnL (DnL StopLf))))) (Lf 7))
:: T TyEmp (B (B L L) (B P L))
```

The term `Up (UpGD (DnL (DnL StopLf)))` is the representation of the pointer $\swarrow 11\uparrow 2$ in de Bruijn notation, which is read from the top as "up and up, then going down (`GD` is short for going down) along the position 11 and stopping at a leaf". The type inference and the type checker automatically ensures well-formedness of cyclic sharing terms.

In Haskell, we can equally use the GADT `T` as an ordinary algebraic datatype, so we can define functions on it by structural recursion as described in Example 2 (even simpler; shape tree and context parameters are unnecessary in defining functions due to Haskell's compilation method [PVWW06]). The implementation and further examples using the GADT `T` are available from the author's home page.

5 General Signature

We give construction of cyclic sharing structures for arbitrary signature as a natural generalisation of the binary tree case.

A *signature* Σ for cyclic sharing structures consists of a set Σ of function symbols having arities. A function symbol of arity $n \in \mathbb{N}$ is denoted by $f^{(n)}$. The set $\mathbb{T}$ of all shape trees is defined by

$$\mathbb{T} \ni \tau ::= \textsc{e} \mid \textsc{p} \mid \textsc{f}(\tau_1, \ldots, \tau_n) \quad \text{for each } f^{(n)} \in \Sigma.$$

Throughout this section, we use the convention that its small capital is associated with each letter of function symbol, e.g. $\textsc{f}$ for f, $\textsc{g}$ for g, etc. The set of all contexts is $\mathbb{T}^* = \{\langle \tau_1, \ldots, \tau_n \rangle \mid n \in \mathbb{N}, i \in \{1, \ldots, n\}, \tau_i \in \mathbb{T}\}$. Positions are defined by $\mathcal{P}os(\textsc{e}) = \mathcal{P}os(\textsc{p}) = \emptyset$, $\mathcal{P}os(\textsc{f}(\tau_1, \ldots, \tau_n)) = \{\epsilon\} \cup \{1.p \mid p \in \mathcal{P}os(\tau_1)\} \cup \ldots \cup \{n.p \mid p \in \mathcal{P}os(\tau_n)\}$.

Typing rules

$$\frac{|\Gamma| = i-1 \quad p \in \mathcal{P}os(\sigma)}{\Gamma, \sigma, \Gamma' \vdash \swarrow p\uparrow i : \textsc{p}} \qquad \frac{\gamma_1, \Gamma \vdash t_1 : \tau_1 \quad \cdots \quad \gamma_n, \Gamma \vdash t_n : \tau_n \quad f^{(n)} \in \Sigma}{\Gamma \vdash f(t_1, \ldots, t_n) : \textsc{f}(\tau_1, \ldots, \tau_n)}$$

where $\gamma_1 = \textsc{f}(\textsc{e}, \ldots, \textsc{e})$, $\gamma_{i+1} = \textsc{f}(\tau_1, \ldots, \tau_i, \textsc{e}, \ldots, \textsc{e})$ for each $1 \le i \le n-1$.

The shape trees γ_i's are also used below. The base category is $(\mathbf{Set}^{\mathbb{T}^*})^{\mathbb{T}}$. For a signature Σ, we associate a signature functor $\Sigma : (\mathbf{Set}^{\mathbb{T}^*})^{\mathbb{T}} \to (\mathbf{Set}^{\mathbb{T}^*})^{\mathbb{T}}$ defined by

$$(\Sigma A)_{\textsc{e}} = 0 \qquad (\Sigma A)_{\textsc{p}} = \mathrm{PO} \qquad (\Sigma A)_{\textsc{f}(\tau_1, \ldots, \tau_n)} = \prod_{1 \le i \le n} \delta_{\gamma_i} A_{\tau_i} \quad \text{for each } f^{(n)} \in \Sigma$$

Theorem 5 (Initial algebra). *Let Σ be a signature. $T_\tau(\Gamma) = \{t \mid \Gamma \vdash t : \tau\}$ forms an initial Σ-algebra where operations are:*

$$\begin{array}{rcl} \mathsf{ptr}^T(\Gamma) : \mathrm{PO}(\Gamma) & \to & T_{\textsc{p}}(\Gamma) \\ \swarrow p\uparrow i & \mapsto & \swarrow p\uparrow i \end{array} \qquad \begin{array}{rcl} f^T(\Gamma) : \prod_{1 \le i \le n} T_{\tau_i}(\gamma_i, \Gamma) & \to & T_{\textsc{f}(\tau_1, \ldots, \tau_n)}(\Gamma) \\ t_1, \ldots, t_n & \mapsto & f(t_1, \ldots, t_n). \end{array}$$

Theorem 6 (Structural recursion). *The unique homomorphism $\phi : T \to A$ is*

$$\phi_{\mathrm{P}}(\Gamma)(\swarrow p{\uparrow}i) = \mathsf{ptr}^A(\Gamma)(\swarrow p{\uparrow}i)$$
$$\phi_{\mathrm{F}(\tau_1,\ldots,\tau_n)}(\Gamma)(f(t_1,\ldots,t_n)) = f^A(\Gamma)(\phi_{\tau_1}(\gamma_1,\Gamma)(t_1),\ldots,\phi_{\tau_n}(\gamma_n,\Gamma)(t_n))$$

Theorem 7 (Structural induction). *To prove that $P^{\Gamma}_{\tau}(t)$ holds for all $t \in T_\tau(\Gamma)$, it suffices to show*

(i) $P^{\Gamma}_{\mathrm{P}}(\swarrow p{\uparrow}i)$ holds for all $\swarrow p{\uparrow}i \in \mathrm{PO}(\Gamma)$,
(ii) if $f^{(n)} \in \Sigma$ and $P^{\gamma_i,\Gamma}_{\tau_i}(t_i)$ holds for all $i = 1,\ldots,n$, then $P^{\Gamma}_{\mathrm{F}(\tau_1,\ldots,\tau_n)}(f(t_1,\ldots,t_n))$ holds.

Moreover, to define a GADT for a given signature is straightforward as we have done in Sect. 4 for the signature of binary cyclic sharing trees.

6 Connection to Equational Term Graphs in the Initial Algebra Framework

We have investigated a term syntax for cyclic sharing structures, which gives a representation of a graph. In this section, we give the converse, i.e., an explicit way to calculate the graph for a given cyclic sharing term. This means to give a semantics of cyclic sharing term by a finite graph. We give it by using Ariola and Klop's equational term graphs in the initial algebra framework. This semantics makes clear connections to existing works on semantics of cyclic sharing structures. We will see it in next section.

Equational term graphs [AK96] are another representation of cyclic sharing structures, which has been used in the formalisation of term graph rewriting. This is a representation of a graph by associating a unique name to each node and by writing down the interconnections through a set of recursive equations. For example, the graph in Figure 4 is represented by an equational term graph

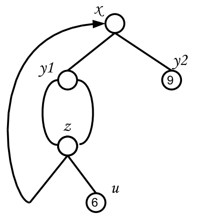

Fig. 4.

$$\{x \mid x = \mathsf{bin}(y_1,y_2), \quad y_2 = \mathsf{lf}(9),$$
$$y_1 = \mathsf{bin}(z,z), \quad z = \mathsf{bin}(x,u),$$
$$u = \mathsf{lf}(6)\}.$$

We use this form of equational term graphs, called flattened form in [AK96] and formally defined as follows (NB. it is slightly different from the original syntax to make explicit the connection to cyclic sharing terms).

Suppose a signature Σ and a set $X = \{x, x_1, \ldots\}$ of variables. *An equational term graph* is of the form $\{x \mid x_1 = t_1,\ x_2 = t_2, \ldots\}$ where each t_i follows the syntax $t ::= x \mid \swarrow p{\uparrow}i \mid f(x_1,\ldots,x_n)$ for each $f^{(n)} \in \Sigma$. A variable is called *bound* if it appears in the left-hand side of an equation, and called *free*, otherwise. We also call $\swarrow p{\uparrow}i$ *free variables* (and regard them as free variables). It is assumed [AK96] that any useless equation $y = t$, where y cannot be reachable from the root, is automatically removed.

We define a translation from a cyclic sharing term to an equational term graph by the unique homomorphism from the initial algebra to an algebra consisting of equational term graphs. The idea is to use positions as the unique variables in an equational term graph. We define $\mathsf{EGraph}_\tau(\Gamma)$ by the set of all equational term graphs having free variables taken from $\mathrm{PO}(\Gamma)$ (where a shape index τ is meaningless for equational term graphs, but we just put this index to form a presheaf). This EGraph form a presheaf in $(\mathbf{Set}^{\mathbb{T}^*})^{\mathbb{T}}$. Any equational term graph can be drawn as a tree-like graph as Fig. 4, hence for each node, we can give its position in the whole equational term graph. So, an equational term graph $\{x_1 \mid x_1 = t_1,\ x_2 = t_2, \ldots\}$ can be normalised to an "α-equivalent form" in which for each $x = t$, the bound variable x is renamed to the position of t in the whole term as $\{\epsilon \mid \epsilon = t'_1,\ 1 = t'_2, \ldots\}$ (see the example below). We identify an equation term graph with its α-normal form.

Proposition 1. EGraph *forms a Σ-algebra, and the unique homomorphism $[\![-]\!] : T \longrightarrow \mathsf{EGraph}$ is monomorphic and gives an interpretation of a cyclic sharing term as a graph represented by an equational term graph.*

Proof. We define an algebra structure on EGraph on "α-equivalent form" as follows.

$$\begin{aligned}
&f_\tau^{\mathsf{EGraph}}(\Gamma)(\{\epsilon \mid G_1\}, \ldots, \{\epsilon \mid G_n\}) = \{\epsilon \mid \epsilon = f(1, \ldots, n),\ G'_1, \ldots, G'_n\}\\
&\qquad \textbf{where } \{1 \mid G'_1\} = \mathsf{shift}_1(\{\epsilon \mid G_1\}) \ \cdots\ \{n \mid G'_n\} = \mathsf{shift}_n(\{\epsilon \mid G_n\})\\
&\mathsf{ptr}_\tau^{\mathsf{EGraph}}(\Gamma)(\swarrow p{\uparrow}i) = \{\epsilon \mid \epsilon = \swarrow p{\uparrow}i\}
\end{aligned}$$

$$\begin{aligned}
&\mathsf{shift}_q\{\epsilon \mid \epsilon = t_1, 1 = t_2, \ldots\} = \{q \mid q = \mathsf{shift}_q(t_1),\ q.1 = \mathsf{shift}_{q.1}(t_2), \ldots\}\\
&\mathsf{shift}_q(x) = q.x \quad \text{for a free variable } x,\\
&\mathsf{shift}_q(f(t_1, \ldots, t_n)) = f(\mathsf{shift}_{q.1}(t_1), \ldots, \mathsf{shift}_{q.n}(t_n))\\
&\mathsf{shift}_q(\swarrow p{\uparrow}i) = \begin{cases} p_n \cdots p_{i+1}\, p \ \textbf{if}\ q \text{ is } p_n \cdots p_{i+1} \cdots p_1,\ |q| \geq i\\ \swarrow p{\uparrow}(i - |q|)\ \textbf{if}\ |q| < i \end{cases}
\end{aligned}$$

The function shift_q shifts every bound variable by a position q (i.e. appending q as prefix) in a term in order to form an equational term graph suitably. Then, it is obvious that $[\![-]\!]$ is monomorphic and gives a translation from cyclic sharing terms to equational term graphs. □

Example 3. Consider the term $\mu x.\mathsf{bin}(\mu y_1.\mathsf{bin}(\mu z.\mathsf{bin}(\uparrow x, \mathsf{lf}(6)), \swarrow 1{\uparrow}y_1), \mathsf{lf}(9))$ of Fig. 2. This is represented as the following term in de Bruijn and is interpreted as an equational term graph:

$$\begin{aligned}
&\mathsf{bin}(\mathsf{bin}(\mathsf{bin}(\uparrow 3, \mathsf{lf}(6)), \swarrow 1{\uparrow}1),\ \mathsf{lf}(9))\\
\overset{[\![-]\!]}{\mapsto}\ &\{\epsilon \mid \epsilon = \mathsf{bin}(1,2), \qquad 12 = 11, \qquad 112 = \mathsf{lf}(6),\\
&\qquad 1 = \mathsf{bin}(11,12), \qquad 111 = \epsilon, \qquad 2 = \mathsf{lf}(9),\\
&\qquad 11 = \mathsf{bin}(111,112)\}.
\end{aligned}$$

7 Further Connections to Other Works

The semantics of cyclic sharing terms by equational term graphs opens connections to other semantics as $T \longrightarrow \mathsf{EGraph} \longrightarrow \mathcal{S}$ where $\mathcal{S}$ is any of the following semantics of equational term graphs.

(i) **letrec**-expressions: an equational term graph is obviously seen as a **letrec**-expression.
(ii) Domain-theoretic semantics: mentioned below.
(iii) Categorical semantics in terms of traced symmetric monoidal categories [Has97].
(iv) Coalgebraic semantics: a graph is seen as a coalgebraic structure that produces every node information along edges, e.g. [AAMV03].

The domain-theoretic semantics of **letrec**-expressions or systems of recursive equations (e.g. [CKV74]), is now standard, which gives infinite expansion of cyclic sharing structures. Via equational term graphs, we can interpret our cyclic sharing terms in each of these semantics. Each semantics has own advantage and principles to reason about some aspects of cyclic sharing structures. However, *none of these has focused on our goals* [I] a simple term syntax that admits structural induction, and [II] direct usability in functional programming, mentioned in Introduction. Hence, we have chosen the initial algebra approach to cyclic sharing structures.

8 Conclusion

We have given an initial algebra characterisation of cyclic sharing structures and derived inductive datatypes, structural recursion and structural induction on them. We have also associated them with equational term graphs in the initial algebra framework, hence we have shown that various ordinary semantics of cyclic sharing structures are equally applied to them.

In programming point of view, practicality of our datatype of cyclic sharing structures still need to be investigated. A possible direction of future work is to use dependently-typed programming language for programming with cyclic sharing structures as an extension of this work.

Acknowledgments. The basis for this work, which was motivated by a question of Zhenjiang Hu, was done while the author visited IPL, University of Tokyo during April 2007 - March 2008. I express my sincere gratitude to Masato Takeichi and members of IPL for the opportunity to stay in a stimulating and pleasant research environment. I am also grateful to Varmo Vene and Tarmo Uustalu for discussion on datatypes in Haskell and reading early draft. This work is supported by the JSPS Grant-in-Aid for Scientific Research (19700006) and NII collaboration research grant. Finally, special thanks to Mika.

References

[AAMV03] P. Aczel, J. Adámek, S. Milius, and J. Velebil. Infinite trees and completely iterative theories: a coalgebraic view. *Theor. Comput. Sci.*, 300(1-3):1–45, 2003.

[AK96] Z. M. Ariola and J. W. Klop. Equational term graph rewriting. *Fundam. Inform.*, 26(3/4):207–240, 1996.

[Bro05] J. Brotherston. Cyclic proofs for first-order logic with inductive definitions. In *Proc. of TABLEAUX'05*, LNAI 3702, pages 78–92, 2005.

[BvEG+87] H.P. Barendregt, M.C.J.D. van Eekelen, J.R.W. Glauert, R. Kennaway, M. J. Plasmeijer, and M. Ronan Sleep. Term graph rewriting. In *Parallel Architechtures and Languages Europe*, LNCS 259, pages 141–158, 1987.

[CFR+91] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, 1991.

[CGZ05] C. Calcagno, P. Gardner, and U. Zarfaty. Context logic and tree update. In *Proc. of POPL'05*, pages 271–282, 2005.

[CKV74] B. Courcelle, G. Kahn, and J. Vuillemin. Algorithmes d'equivalence et de reduction a des expressions minimales dans une classe d'equations recursives simples. In *ICALP*, pages 200–213, 1974.

[Dyb94] P. Dybjer. Inductive families. *Formal Aspects of Computing*, 6:440–465, 1994.

[Fio02] M. Fiore. Semantic analysis of normalisation by evaluation for typed lambda calculus. In *Proc. of PPDP'02*, pages 26–37, 2002.

[Fio08] M. Fiore. Second-order and dependently-sorted abstract syntax. In *Proc. of LICS'08*, pages 57–68, 2008.

[FPT99] M. Fiore, G. Plotkin, and D. Turi. Abstract syntax and variable binding. In *Proc. of 14th Annual Symposium on Logic in Computer Science*, pages 193–202, 1999.

[GHUV06] N. Ghani, M. Hamana, T. Uustalu, and V. Vene. Representing cyclic structures as nested datatypes. In *Proc. of TFP'06*, pages 173–188, 2006.

[GJ07] N. Ghani and P. Johann. Initial algebra semantics is enough! In *Proc. of TLCA'07*, LNCS 4583, pages 207–222, 2007.

[GUH06] N. Ghani, T. Uustalu, and M. Hamana. Explicit substitutions and higher-order syntax. *Higher-Order and Symbolic Computation*, 19(2/3):263–282, 2006.

[Ham04] M. Hamana. Free Σ-monoids: A higher-order syntax with metavariables. In *Proc. of APLAS'04*, LNCS 3302, pages 348–363, 2004.

[Has97] M. Hasegawa. *Models of Sharing Graphs: A Categorical Semantics of let and letrec.* PhD thesis, University of Edinburgh, 1997.

[Has02] R. Hasegawa. Two applications of analytic functors. *Theor. Comput. Sci.*, 272(1-2):113–175, 2002.

[HJ98] C. Hermida and B. Jacobs. Structural induction and coinduction in a fibrational setting. *Inf. Comput.*, 145(2):107–152, 1998.

[JG08] P. Johann and N. Ghani. Foundations for structured programming with GADTs. In *Proc. of POPL'08*, pages 297–308, 2008.

[McB02] C. McBride. Faking it: Simulating dependent types in Haskell. *J. Funct. Program.*, 12(4&5):375–392, 2002.

[MS03] M. Miculan and I. Scagnetto. A framework for typed HOAS and semantics. In *Proc. of PPDP'03*, pages 184–194. ACM Press, 2003.

[Nor07] U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007.

[PVWW06] S. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for GADTs. In *Proc. of ICFP '06*, pages 50–61, 2006.

[Rob02] E. Robinson. Variations on algebra: monadicity and generalisations of equational theories. *Formal Aspects of Computing*, 13(3-5):308–326, 2002.

[SP82] M. B. Smyth and G. D. Plotkin. The category-theoretic solution of recursive domain equations. *SIAM J. Comput*, 11(4):763–783, 1982.

[Tar72] R.E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal of Computing*, 1(2):146–160, 1972.