Strongly Normalising Cyclic Data Computation by Iteration Categories of Second-Order Algebraic Theories

Makoto Hamana

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

Abstract

Cyclic data structures, such as cyclic lists, in functional programming are tricky to handle because of their cyclicity. This paper presents an investigation of categorical, algebraic, and computational foundations of cyclic datatypes. Our framework of cyclic datatypes is based on second-order algebraic theories of Fiore et al., which give a uniform setting for syntax, types, and computation rules for describing and reasoning about cyclic datatypes. We extract the "fold" computation rules from the categorical semantics based on iteration categories of Bloom and Esik. Thereby, the rules are correct by construction. Finally, we prove strong normalisation using the General Schema criterion for second-order computation rules. Rather than the fixed point law, we particularly choose Bekič law for computation, which is a key to obtaining strong normalisation.

1998 ACM Subject Classification D.3.2 Language Classifications, E.1 Data Structures, F.3.2 Semantics of Programming Languages

Keywords and phrases cyclic data structures, traced cartesian category, fixed point, functional programming, fold

Digital Object Identifier 10.4230/LIPIcs.FSCD.2016.21

1 Introduction

Cyclic data structures in functional programming are tricky to handle because of their cyclicity. In Haskell, one can define cyclic data structures, such as cyclic lists by

```
clist = 2:1:clist
```

The feasibility of such a recursive definition of cyclic data depends on lazy evaluation. However, it does not ensure termination of computation. It might fall into a non-terminating situation. For example, what is the sum of all elements of `clist`? One may think that it is non-terminating, undefined, or impossible.

An answer using our framework in this paper is different. We do not rely on lazy evaluation. We provide a way to regard the sum of a cyclic list as a cyclic natural number, which is computed by the strongly normalising "fold" combinator. In this paper, we investigate a framework for syntax and semantics of *cyclic datatypes* that makes this understanding and computation correct.

Our framework of cyclic datatypes is founded on second-order algebraic theories of Fiore et al. [13, 14]. Second-order algebraic theories have been shown to be a useful framework that models various important notions of programming languages, such as logic programming [32], algebraic effects [15], quantum computation [33]. This paper gives another application of second-order algebraic theories, namely, to cyclic datatypes and its computation. We use second-order algebraic theories to give a uniform setting for typed syntax, equational

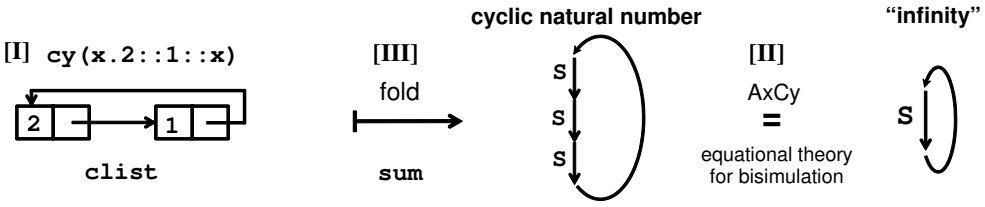


Figure 1 Framework: Second-order algebraic theories and iteration theories.

logic and computation rules for describing and reasoning about cyclic datatypes. We extract computation rules for the fold from the categorical semantics based on iteration categories [5]. Thereby the rules are correct by construction. Finally, we prove strong normalisation by using the General Schema criterion [3] for rewrite rules.

Overview. As an overview of cyclic datatypes and their operations we develop in this paper, we first demonstrate descriptions and an operation of cyclic datatypes by pseudo-program codes. The code fragments correspond one-to-one to theoretical data given in later sections. Therefore, they are theoretically meaningful and more intuitive than starting from detailed theory.

First we consider an example of cyclic lists. The code below with the keyword `ctype` is intended to declare cyclic datatype `CList` of cyclic lists having two ordinary constructors in Haskell or Agda style.

```
ctype CList where
  [] : CList
  :: : CNat,CList → CList
with axioms AxCy
```

We assume that any `ctype` declared datatype has a default constructor "`cy`" for making a cycle. For example, we express a cyclic list of 1 as a term `cy(x.1::x)`, where `cy` has a variable binding "`x.`", regarded as the "address" of the top of list. A variable occurrence `x` in the body means to refer to the top, hence it makes a cycle. The terms built from the constructors of `CList` and the default constructor `cy` is required to satisfy the axioms AxCy (given later in Fig. 3) as the keyword "`with axioms`" mentioned (we assume that any `ctype` datatype satisfies AxCy, so this is for ease of understanding). We next consider the above mentioned example of the sum of cyclic list. We define another cyclic datatype of natural numbers.

```
ctype CNat where
  0 : CNat
  S : CNat → CNat
with axioms AxCy
```

```
sum : CList → CNat
spec sum ([])    = 0
     sum (k::t) = plus(k, sum (t))
```

The code with keyword `spec` at the right column describes an equational *specification* of function. It requires that the sum function from cyclic lists to cyclic natural numbers must satisfy the ordinary definition. We intend that the `spec` code is merely a (loose) specification, and not a definition, because it lacks the case of `cy`-term.

```
fun sum t = fold (0, k,x.plus(k,x)) t
```

We here assume that the `plus` function on `CNat` has already been defined (as presented later in Example 4.2). The above code with the keyword `fun` defines the function `sum`. It is defined

by the `fold` combinator on the cyclic datatypes, as in an ordinary fold on an algebraic datatypes. The first two arguments `0` and `k,x.plus(k,x)` correspond to the right-hand sides of the specification of `sum`. The `fold` is actually the fold on a cyclic datatype, which knows how to cope with `cy`-term. Actually, the sum of a cyclic list can be computed as follows:

$$\texttt{sum(cy(x.S}^2\texttt{(0)::S(0)::x))} \to$$
$$\texttt{cy(x.sum(x.S}^2\texttt{(0)::S(0)::x)@x)} \to^+ \texttt{cy(x.S(S(S(x))))}$$

where we represent n by $\texttt{S}^n(0)$. The final term is a normal form that cannot be rewritten further. Therefore, we regard it as the computation result. Here `sum` is intended to denote "`fold`(0,..)". The steps presented above are actual rewrite steps by the second-order rewrite rules FOLDr given later in Fig. 8.

How to understand the meaning of the result `cy(x.S(S(S(x))))` is arguable. The overall situation we have demonstrated is illustrated in Fig. 1. In this paper, we also provide a formal basis to understand and to reason about cyclic data, as well as the computation result. We use second-order equational logic and the axioms AxCy to equate cyclic data formally (Fig. 1 [III]). It completely characterises the notion of bisimulation on cyclic data. The expression `cy(x.S(S(S(x))))` is equal to (i.e. bisimilar to) `cy(x.S(x))`, which is a minimal representation of the result, which may be regarded as ∞ (infinity). In this paper, we do not develop an explicit algorithm to extract such a minimal representation from the computation result, but it is noteworthy that this equational theory generated by AxCy is decidable. Consequently, it is computationally reasonable. More practical examples on cyclic datatypes and computation will be given in §6.

2 Second-Order Algebraic Theory of Cyclic Datatypes

We introduce the framework of second-order cartesian algebraic theory, which is a typed and cartesian extension of [13, 14] and [19]. Here "cartesian" means that the target sort of a function symbol is a sequence of base types. We use second-order algebraic theory as a formal framework to provide syntax and to describe axioms of algebraic datatypes enriched with cyclic constructs. The second-order feature is necessary for cycle operation and the fold function on them. We will often omit superscripts or subscripts of a mathematical object if they are clear from contexts. We use the vector notation $\overrightarrow{A}$ for a sequence $A_1, \cdots, A_n$, and $|\overrightarrow{A}|$ for its length.

2.1 Cartesian Second-Order Algebraic Theory

We assume that $\mathcal{B}$ is a set of *base types* (written as $a, b, c, \ldots$), and Σ, called a *signature*, is a set of function symbols of the form

$$f : (\overrightarrow{a_1} \to \overrightarrow{b_1}), \ldots, (\overrightarrow{a_m} \to \overrightarrow{b_m}) \to c_1, \ldots, c_n.$$

where all a_i, b_i, c_i are base types (thus any function symbol is of up to second-order type). A sequence of types may be empty in the above definition. The empty sequence is denoted by $()$, which may be omitted, e.g., $b_1, \ldots, b_m \to c$, or $() \to c$. The latter case is simply denoted by c. A signature Σ_c for type c denotes a subset of Σ, where every function symbol is of the form $f : \tau \to c$, which is regarded as a constructor of c. A *metavariable* is a variable of (at most) first-order type, declared as $\mathrm{M} : \overrightarrow{a} \to b$ (written as small-caps letters $\mathrm{Z}, \mathrm{T}, \mathrm{S}, \mathrm{M}, \ldots$). A *variable* of the order 0 type is merely called variable (written usually $x, y, \ldots$). The raw syntax is given as follows.

- *Terms* have the form: $t ::= x \mid x.t \mid f(t_1, \ldots, t_n)$.
- *Meta-terms* extend terms to: $t ::= x \mid x.t \mid f(t_1, \ldots, t_n) \mid \mathrm{M}[t_1, \ldots, t_n]$.

Terms are used for representing concrete cyclic data, functional programs on them and equations we want to model. A second-order equational theory is a set of proved equations built from terms (NB. not meta-terms). Meta-terms are used for formulating equational axioms, which are expected to be instantiated to terms. We write $x_1, \ldots, x_n.\, t$ for $x_1.\cdots.x_n.\, t$.

A metavariable context Θ is a sequence of metavariable:type-pairs, and a context Γ is a sequence of variable:type-pairs. A judgment is of the form $\Theta \triangleright \Gamma \vdash t : \overrightarrow{b}$. If Θ is empty, we may simply write $\Gamma \vdash t : \overrightarrow{b}$. A meta-term t is well-typed by the typing rules Fig. 4. We often omit the types for binders as $f(\, \overrightarrow{x_1}.t_1, \ \ldots, \ \overrightarrow{x_n}.t_n \,)$. Given a meta-term t with free variables $x_1, \ldots, x_n$, the notation $s\,[x_1 \mapsto s_1, \ldots, x_n \mapsto s_n]$ denotes ordinary capture avoiding substitution that replaces the variables with meta-terms $s_1, \ldots, s_n$. A substitution which replaces metavariables with meta-terms [14] is defined in Fig. 6. For meta-terms $\Theta \triangleright \Gamma \vdash s : \overrightarrow{b}$ and $\Theta \triangleright \Gamma \vdash t : \overrightarrow{b}$, an *equation* is of the form $\Theta \triangleright \Gamma \vdash s = t : \overrightarrow{b}$, or denoted by $\Gamma \vdash s = t : \overrightarrow{b}$ when Θ is empty. The cartesian second-order equational logic is a logic to deduce formally proved equations. The inference system of equational logic is given in Fig. 5.

Preliminaries for datatypes. The *default signature* Σ_{def} is given by the function symbols called *default constructors*:

Empty sequence	$\langle\,\rangle : ()$	Tuple	$\langle -, \cdots, - \rangle : (\overrightarrow{c_1}), \ldots, (\overrightarrow{c_n}) \to \overrightarrow{c_1}, \ldots, \overrightarrow{c_n}$
Cycle constr.	$\mathsf{cy}^{\lvert\overrightarrow{c}\rvert} : (\overrightarrow{c} \to \overrightarrow{c}) \to \overrightarrow{c}$	Composition	$\diamond_{(\overrightarrow{a},\overrightarrow{c})} : (\overrightarrow{a} \to \overrightarrow{c}), \overrightarrow{a} \to \overrightarrow{c}$.

defined for all base types $\overrightarrow{a}, \overrightarrow{c}, \overrightarrow{c_1}, \ldots, \overrightarrow{c_n} \in \mathcal{B}$. This means that any base type has default constructors. We assume that any signature includes Σ_{def} in this paper. We identify $\langle t, \langle\,\rangle\rangle$ and $\langle \langle\,\rangle, t\rangle$ with t, and $\langle \langle s\,,\, t\rangle\,,\, u\rangle$ with $\langle s\,,\, \langle t\,,\, u\rangle\rangle$; thus we will freely omit the angle brackets as $\langle t_1\,,\, \ldots\,,\, t_n\rangle$.

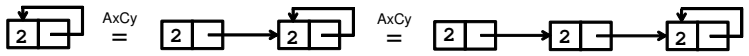
A *datatype declaration* for a type c is given by a triple $(c, \Sigma_c, \mathcal{E})$ consisting of a base type c, signature Σ and equational axioms $\mathcal{E}$, where every $f \in \Sigma_c$ is first-order, i.e. is of the form $f : b_1, \ldots, b_n \to c$, and any equation in $\mathcal{E}$ is built from Σ_c-terms.

2.2 Instance (1): Cyclic Lists modulo Bisimulation

We will present an algebraic formulation of cyclic datatypes. By cyclic datatype, we mean algebraic datatype having the cycle construct `cy` satisfying the axioms that characterise cyclicity. The first example is the datatype of cyclic lists. It has already been defined as `CList` in Introduction as the pseudo code. We now give a formal definition using datatype declaration. Fix $a \in \mathcal{B}$. The datatype declaration for $\mathtt{CList}_a$, the cyclic lists of type a, is given by $(\mathtt{CList}_a, \Sigma_{\mathtt{CList}_a}, \mathsf{AxCy})$ where $\Sigma_{\mathtt{CList}_a}$ is

$$[\,] : \mathtt{CList}_a, \qquad (- ::_a -) : a, \mathtt{CList}_a \to \mathtt{CList}_a$$

and the axioms AxCy are given in Fig. 3. Note that $\mathtt{CList}_a$ has also the default constructors, thus one can form cycle (see the example below). The definition of `CList` in Introduction actually represents the datatype declaration $(\mathtt{CList}_{\mathtt{CNat}}, \Sigma_{\mathtt{CList}_{\mathtt{CNat}}}, \mathsf{AxCy})$. Hereafter, we will omit the type parameter subscript a of `CList`. The axioms AxCy mathematically characterise that `cy` is truly a cycle constructor in the sense of Conway fixed point operator [5]. The equational theory generated by AxCy captures the intended meaning of cyclic lists. For example, the following are identified as the same cyclic list:



$$\mathsf{cy}(x.2 :: x) = \quad 2 :: \mathsf{cy}(x.2 :: x) \quad = \quad 2 :: 2 :: \mathsf{cy}(x.2 :: x)\,.$$

These equalities come from the *fixed point property* of cy.

On axioms AxCy We explain the intuitive meaning of the axioms in AxCy. Parameterised fixed-point axioms axiomatise cy as a fixed point operator. They (minus (CI)) are equivalent to the axioms for Conway operators of [5, 23, 31]. Bekič law is well-known in denotational semantics (cf. [34, §10.1]) to calculate the fixed point of a pair of continuous functions. Conway operators are also arisen in work independently of Hyland and Hasegawa [23], who established a connection with the notion of traced cartesian categories [25]. There are equalities that holds in the cpo semantics but Conway operators do not satisfy. The axiom (CI) is the commutative identities of Bloom and Ésik [5, 31], which ensures that all equalities that hold in the cpo semantics do hold. See also [31, Section 2] for a useful overview about this. The equality generated by AxCy is actually bisimulation on cyclic lists. This is included in the equality of cyclic sharing trees given in next subsection.

2.3 Instance (2): Cyclic Sharing Trees modulo Bisimulation

Next we consider the datatype of binary branching trees, which can involve cycle and sharing. We call them *cyclic sharing trees*, or simply cyclic trees. We first give the declaration of datatype `CTree` of cyclic trees as the style of pseudo code below, where we assume that f's

```
ctype CTree where
  f : CTree → CTree
  [] : CTree
  + : CTree,CTree → CTree
with axioms AxCy,AxBr([],+)
```

part denotes various unary function symbols such as `a,b,c,p,q,...`. Formally, it is expressed as the datatype declaration ($\mathtt{CTree}$, $\{f, [], +\}$, $\mathsf{AxCy} \cup \mathsf{AxBr}([], +)$). The binary operator `+` denotes a branch. For example, one can write `b([])+c([])` (cf. Fig. 2 (A)). It can also express sharing by the constructor $\diamond$ of composition: `x.a(b(x) + c(x)) ◇ p([])` (Fig. 2 (F)). Note that the first argument of composition $\diamond$ has a binder (e.g. `x.`), which indicates placeholder filled by the shared part after $\diamond$ (e.g. `p([])`). A binder at the first argument of $\diamond$-term may be a sequence of variables (e.g. "`x,y.`" in (E)), which will be filled by terms in a tuple (e.g. `<p([]),q([])>`). Cyclic trees are very expressive. They cover essentially XML trees with IDREF, the data model called *trees with pointers* [7], and arbitrary rooted directed graphs (cf. Fig. 2 (B)(E)).

We denote by $\sim$ the equivalence relation generated by the axioms $\mathsf{AxCy}, \mathsf{AxBr}([], +)$ in Fig. 3. Using the axioms $\mathsf{AxCy} \cup \mathsf{AxBr}([], +)$, we can reason this equality $\sim$ in the second-order equational logic. The equality $\sim$ gives reasonable meaning of cycles as in the case of cyclic lists and that the branch $+$ is associative, commutative and idempotent (cf. Fig. 2 (C)), thus nested $+$ can be seen as an n-ary branch (cf. Fig. 2 (D)). Moreover, a shared term and its unfolding are also identified by $\sim$ because of the axiom (sub) (cf. Fig. 2 (F)). The axiom (sub) is similar to the β-reduction in the λ-calculus.

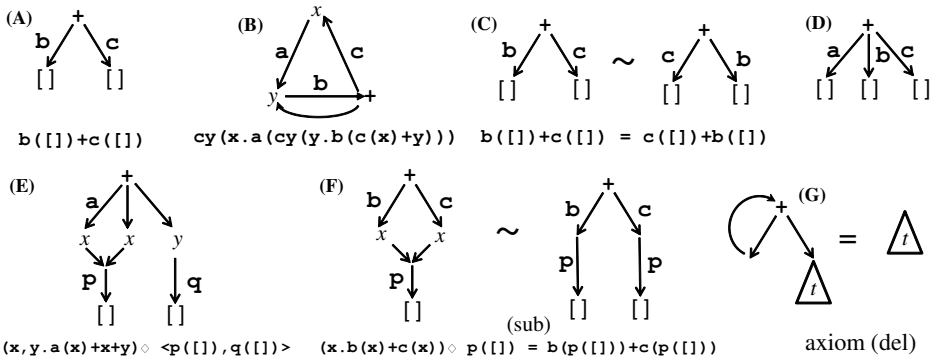


Figure 2 Examples of cyclic sharing trees.

Algebraic theory of bisimulation. Actually, $\sim$ is exactly *bisimulation* on cyclic trees. Since unary f expresses a labelled edge, and $+$ expresses a branch, cyclic sharing trees are essentially process graphs of regular behaviors, called *charts* by Milner in [29]. Infinite unfolding of them are synchronization trees [5]. Thus the standard notion of bisimulation between graphs can be defined. Intuitively, starting from the root, bisimulation is by comparing traces of labels of two graphs along edges (more detailed definition is given in [5, 29] or ([22] Appendix)). Now we see that (C),(F) and (G) are examples of bisimulation. Actually, the axioms in Fig. 3 are sound for bisimulation, i.e. for each axiom, the left and the right-hand sides are bisimilar. Moreover, it is complete.

▶ **Proposition 2.1** ([21],([22]§5.3)). $\Gamma \vdash s = t \ : \mathtt{CTree}$ *is derivable from* AxCy *and* $\mathsf{AxBr}([\,],+)$ *iff if* s *and* t *are bisimilar.*

The main reason of this is that the axioms AxCy and $\mathsf{AxBr}([\,],+)$ are second-order representation of Bloom and Ésik's complete equational axioms of bisimulation [5]. A crucial fact is that bisimulation $s \sim t$ is decidable [5, 6]. There is also an efficient algorithm for checking bisimulation, e.g. [10]. Hence, cyclic datatypes with the axioms AxCy, or the axioms $\mathsf{AxCy} \cup \mathsf{AxBr}$ are computationally feasible, such as checking equality on cyclic structures we have seen in Fig. 1.

There are many other instances of cyclic datatypes, some of which will be given in §6.

3 Categorical Semantics of Cyclic Datatypes

In this section, we give a categorical semantics of cyclic datatypes. A reason to consider categorical semantics is to systematically obtain a "structure preserving map" on cyclic datatypes. We will formulate the "fold" function for a cyclic datatype as a functor on the category for cyclic datatypes (Thm. 3.8 and §4).

Since a cyclic datatype has cycles, the target categorical structure should have a notion of fixed point. It has been studied in iteration theories of Bloom and Ésik [5]. In particular iteration categories [11] are suitable for our purpose, which are traced cartesian categories [25, 23] satisfying the commutative identities axiom [5]. We write $\mathbf{1}$ for the terminal object, $\times$ for the cartesian product, $\langle -, - \rangle$ for pairing, and $\Delta = \langle \mathrm{id}, \mathrm{id} \rangle$ for diagonal in a cartesian category.

Axioms AxCy for cycles

$$\text{(sub)}\quad \mathrm{T}:\overrightarrow{a}\to\overrightarrow{c},\ \mathrm{S}_1,\dots,\mathrm{S}_n:\overrightarrow{a} \ \triangleright\vdash\ (\overrightarrow{y}.\mathrm{T}[\overrightarrow{y}])\diamond\langle \mathrm{S}_1\,,\,\dots\,,\,\mathrm{S}_n\rangle = \mathrm{T}[\mathrm{S}_1\,,\,\dots\,,\,\mathrm{S}_n] \quad :\overrightarrow{c}$$

$$\text{(SP)}\quad \mathrm{T}:\overrightarrow{c} \ \triangleright\vdash \langle(\overrightarrow{y}.y_1)\diamond\mathrm{T},\dots,(\overrightarrow{y}.y_n)\diamond\mathrm{T}\rangle = \mathrm{T} \quad :\overrightarrow{c}$$

$$\text{(dinat)}\quad \mathrm{S}:\overrightarrow{a}\to\overrightarrow{c},\ \mathrm{T}:\overrightarrow{c}\to\overrightarrow{a} \ \triangleright\vdash\ \mathsf{cy}(\overrightarrow{x}.\mathrm{S}[\mathrm{T}[\overrightarrow{x}]]) = (\overrightarrow{z}.\mathrm{S}[\overrightarrow{z}])\diamond\mathsf{cy}(\overrightarrow{z}.\mathrm{T}[\mathrm{S}[\overrightarrow{z}]]) \quad :\overrightarrow{c}$$

$$\text{(Bekič)}\quad \begin{matrix}\mathrm{T}:\overrightarrow{c},\overrightarrow{a}\to\overrightarrow{c},\\ \mathrm{S}:\overrightarrow{c},\overrightarrow{a}\to\overrightarrow{a}\end{matrix} \ \triangleright\vdash \mathsf{cy}(\overrightarrow{x},\overrightarrow{y}.\langle\widehat{\mathrm{T}},\widehat{\mathrm{S}}\rangle) = \begin{matrix}\langle\ \mathsf{cy}(\overrightarrow{x}.(\overrightarrow{y}.\widehat{\mathrm{T}})\diamond\mathsf{cy}(\overrightarrow{y}.\widehat{\mathrm{S}})),\\ \ \ \mathsf{cy}(\overrightarrow{y}.(\overrightarrow{x}.\widehat{\mathrm{S}})\diamond\mathsf{cy}(\overrightarrow{x}.(\overrightarrow{y}.\widehat{\mathrm{T}})\diamond\mathsf{cy}(\overrightarrow{y}.\widehat{\mathrm{S}})))\ \rangle\end{matrix} :\overrightarrow{c},\overrightarrow{a}$$

$$\text{(CI)}\quad \mathrm{T}:\overrightarrow{a}\to\overrightarrow{a} \ \triangleright\vdash\ \mathsf{cy}(\overrightarrow{y}.\langle\mathrm{T}[\rho_1],\dots,\mathrm{T}[\rho_m]\rangle) = \langle\mathsf{cy}(y.\tilde{\mathrm{T}}),\dots,\mathsf{cy}(y.\tilde{\mathrm{T}})\rangle \quad :\overrightarrow{a}$$

In (CI), $\rho_i = \langle q_1\,,\,\dots\,,\,q_m\rangle$, each q_j is one of y_i for $i = 1,\dots,m$, and $\tilde{\mathrm{T}}$ is short for $\mathrm{T}[y,\dots,y]$.
In (Bekič), $\widehat{\mathrm{T}}$ and $\widehat{\mathrm{S}}$ are short for $\mathrm{T}[\overrightarrow{x},\overrightarrow{y}]$ and $\mathrm{S}[\overrightarrow{x},\overrightarrow{y}]$, respectively.

Axioms AxBr($[\,],+$) for branching

$$\begin{array}{llllllll}
\text{(del)} & \mathrm{T}:c & \triangleright\vdash & \mathsf{cy}(x^c.\mathrm{T}+x) & = & \mathrm{T} & :c\\
\text{(unitL)} & \mathrm{T}:c & \triangleright\vdash & [\,]+\mathrm{T} & = & \mathrm{T} & :c\\
\text{(unitR)} & \mathrm{T}:c & \triangleright\vdash & \mathrm{T}+[\,] & = & \mathrm{T} & :c\\
\text{(assoc)} & \mathrm{S},\mathrm{T},\mathrm{U}:c & \triangleright\vdash & (\mathrm{S}+\mathrm{T})+\mathrm{U} & = & \mathrm{S}+(\mathrm{T}+\mathrm{U}) & :c\\
\text{(comm)} & \mathrm{S},\mathrm{T}:c & \triangleright\vdash & \mathrm{S}+\mathrm{T} & = & \mathrm{T}+\mathrm{S} & :c\\
\text{(degen)} & \mathrm{T}:c & \triangleright\vdash & \mathrm{T}+\mathrm{T} & = & \mathrm{T} & :c
\end{array}$$

Figure 3 Axioms.

▶ **Definition 3.1** ([11, 5]). A *Conway operator* in a cartesian category $\mathcal{C}$ is a family of functions $(-)^\dagger : \mathcal{C}(A\times X, X)\to\mathcal{C}(A,X)$ satisfying:

$$(f\circ(g\times\mathrm{id}_X))^\dagger = f^\dagger\circ g\ ,\qquad (f^\dagger)^\dagger = (f\circ(\mathrm{id}_A\times\Delta))^\dagger\ ,$$
$$f\circ\langle\mathrm{id}_A,(g\circ\langle\pi_1,f\rangle)^\dagger\rangle = (f\circ\langle\pi_1,g\rangle)^\dagger.$$

An *iteration category* is a cartesian category having a Conway operator satisfying the "commutative identities" law [5]

$$\langle f\circ(\mathrm{id}_A\times\rho_1),\dots,f\circ(\mathrm{id}_A\times\rho_m)\rangle^\dagger = \Delta_m\circ(f\circ(\mathrm{id}_A\times\Delta_m))^\dagger : A\to X$$

where

- $f : A\times X^m\to X$
- diagonal $\Delta_m \triangleq \langle\mathrm{id}_X,\cdots,\mathrm{id}_X\rangle : X\to X^m$
- $\rho_i : X^m\to X^m$ such that $\rho_i = \langle q_{i1},\dots,q_{im}\rangle$ where each q_{ij} is one of projections $\pi_1,\dots,\pi_m : X^m\to X$ for $i = 1,\dots,m$ (see also [31]).

An *iteration functor* between iteration categories is a cartesian functor that preserves Conway operators.

A typical example of iteration category is the category of complete partial orders (cpos) and continuous functions [5, 23], where the least fixed point operator is a Conway operator.

▶ **Definition 3.2.** Let Σ be a signature. A Σ-*structure* M in an iteration category $\mathcal{C}$ is specified by giving for each base type $b\in\mathcal{B}$, an object $[\![b]\!]^M$ (or simply written $[\![b]\!]$) in $\mathcal{C}$, and for each function symbol $f : (\overrightarrow{a_1}\to\overrightarrow{b_1}),\dots,(\overrightarrow{a_m}\to\overrightarrow{b_m})\to\overrightarrow{c}$, a function

$$[\![f]\!]^M_A : \mathcal{C}(A\times[\![\overrightarrow{a_1}]\!],[\![\overrightarrow{b_1}]\!])\times\cdots\times\mathcal{C}(A\times[\![\overrightarrow{a_n}]\!],[\![\overrightarrow{b_n}]\!])\longrightarrow\mathcal{C}(A,[\![\overrightarrow{c}]\!]) \tag{1}$$

$$\frac{y : b \in \Gamma}{\Theta \rhd \Gamma \vdash\ y\ : b} \qquad \frac{(\mathrm{M} : a_1, \ldots, a_m \to \overrightarrow{b}) \in \Theta \quad \Theta \rhd \Gamma \vdash\ t_1\ : a_1 \quad \Theta \rhd \Gamma \vdash\ t_m\ : a_m}{\Theta \rhd \Gamma \vdash\ \mathrm{M}[t_1, \ldots, t_m]\ : \overrightarrow{b}}$$

$$\frac{\Theta \rhd \Gamma, \overrightarrow{x_1 : a_1} \vdash\ t_1\ : \overrightarrow{b_1} \quad \cdots \quad \Theta \rhd \Gamma, \overrightarrow{x_m : a_m} \vdash\ t_m\ : \overrightarrow{b_m}}{\Theta \rhd \Gamma \vdash\ f(\ \overrightarrow{x_1^{a_1}}.t_1, \ldots \overrightarrow{x_m^{a_m}}.t_m\ )\ : \overrightarrow{c}}$$

where $f : (\overrightarrow{a_1} \to \overrightarrow{b_1}), \ldots, (\overrightarrow{a_m} \to \overrightarrow{b_m}) \to \overrightarrow{c}$.

Figure 4 Typing rules of meta-terms.

(Sub)
$$\frac{\mathrm{M}_1 : (\overrightarrow{a_1} \to \overrightarrow{b_1}), \ldots, \mathrm{M}_k : (\overrightarrow{a_k} \to \overrightarrow{b_k}) \rhd \Gamma \vdash\ t = t'\ : \overrightarrow{c} \quad \Theta \rhd \Gamma', \overrightarrow{x_i : a_i} \vdash\ s_i = s_i'\ : \overrightarrow{b_i} \quad (1 \le i \le k)}{\Theta \rhd \Gamma, \Gamma' \vdash\ t\,[\,\overrightarrow{\mathrm{M} := s}\,] = t'\,[\,\overrightarrow{\mathrm{M} := s'}\,]\ : \overrightarrow{c}}$$

(Ax)
$$\frac{(\Theta \rhd \Gamma \vdash\ s = t\ : \overrightarrow{c}) \in \mathcal{E}}{\Theta \rhd \Gamma \vdash\ s = t\ : \overrightarrow{c}}$$

(Ref)
$$\frac{}{\Theta \rhd \Gamma \vdash\ t = t\ : \overrightarrow{c}}$$

(Sym)
$$\frac{\Theta \rhd \Gamma \vdash\ s = t\ : \overrightarrow{c}}{\Theta \rhd \Gamma \vdash\ t = s\ : \overrightarrow{c}}$$

(Tr)
$$\frac{\Theta \rhd \Gamma \vdash\ s = t\ : \overrightarrow{c} \quad \Theta \rhd \Gamma \vdash\ t = u\ : \overrightarrow{c}}{\Theta \rhd \Gamma \vdash\ s = u\ : \overrightarrow{c}}$$

Figure 5 Cartesian second-order equational logic.

which is natural in A, where $[\![b_1, \ldots, b_n]\!] \triangleq [\![b_1]\!] \times \ldots \times [\![b_n]\!]$. Also given a context $\Gamma = x_1 : b_1, \ldots, x_n : b_n$ we set $[\![\Gamma]\!] \triangleq [\![b_1, \ldots, b_n]\!]$. The superscript of $[\![-]\!]$ may be omitted hereafter.

Interpretation. Let M be a Σ-structure in an iteration category $\mathcal{C}$. We give the categorical meaning of a term judgment $\Gamma \vdash t\ : \overrightarrow{c}$ (where there are no metavariables) as a morphism $[\![t]\!]^M : [\![\Gamma]\!] \to [\![\overrightarrow{c}]\!]$ in $\mathcal{C}$ defined by

$$[\![\Gamma \vdash y_i\ : c]\!]^M = \pi_i : [\![\Gamma]\!] \to [\![c]\!]$$
$$[\![\Gamma \vdash f(\ \overrightarrow{x_1^{a_1}}.t_1, \ldots \overrightarrow{x_n^{a_n}}.t_n\ )\ : \overrightarrow{c}]\!]^M = [\![f]\!]^M_{[\![\Gamma]\!]}(\ [\![t_1]\!]^M, \ldots, [\![t_n]\!]^M\ ).$$

We assume the following interpretations in any Σ_{def}-structure:

$$[\![\langle\,\rangle]\!]^M_A = !\ : A \to \mathbf{1} \qquad [\![\langle -, \ldots, - \rangle]\!]^M_A(t_1, \ldots, t_n) = \langle t_1, \ldots, t_n \rangle$$
$$[\![\diamond]\!]^M_A(t, s) = t \circ \langle \mathrm{id}_A, s \rangle \qquad [\![\mathsf{cy}]\!]^M_A(t) = t^\dagger$$

Importantly, these satisfy the axioms AxCy because $\mathcal{C}$ is an iteration category.

▶ **Definition 3.3.** A $(\Sigma, \mathcal{E})$-*structure* is a Σ-structure M in $\mathcal{C}$ satisfying $[\![s]\!]^M = [\![t]\!]^M$ for every axiom $\Gamma \vdash s = t\ : c$ in $\mathcal{E}$. Let N be a $(\Sigma, \mathcal{E})$-structure in an iteration category $\mathcal{D}$. We say that an iteration functor $F : \mathcal{C} \to \mathcal{D}$ *preserves* $(\Sigma, \mathcal{E})$-*structures* if $F([\![-]\!]^M) = [\![-]\!]^N$.

A c-*structure* (M, α) for a datatype declaration $(c, \Sigma_c, \mathcal{E})$ is a $(\Sigma_c, \mathcal{E})$-structure M with a family of morphisms of $\mathcal{C}$; $\alpha \triangleq (\ (\!|f|\!)^M : [\![b_1]\!] \times \ldots \times [\![b_n]\!] \to [\![c]\!]\ )_{f : b_1, \ldots, b_n \to c \in \Sigma_c}$. It

Let $\Gamma = y_1 : \overrightarrow{b_1} \cdots, y_k : \overrightarrow{b_1}$. Suppose $\Theta \triangleright \Gamma', \overrightarrow{x_i : a_i} \vdash \ \ s_i \ : \overrightarrow{b_i}$ and

$$\mathrm{M}_1 : \overrightarrow{a}_1 \to \overrightarrow{b}_1, \ldots, \mathrm{M}_k : \overrightarrow{a}_k \to \overrightarrow{b}_k \triangleright \Gamma \vdash \ e \ : \overrightarrow{c}$$

where $\overrightarrow{x_i : a_i} = x_i^1 : a_i^1, \ldots, x_i^m : a_i^m$ for $m = |\overrightarrow{a_i}|$ and each $i = 1, \ldots, k$. Then a substitution $\Theta \triangleright \Gamma, \Gamma' \vdash \ e\,[\overrightarrow{\mathrm{M} := s}] \ : \overrightarrow{c}$ is inductively defined as follows.

$$\begin{array}{rcl} x\,[\overrightarrow{\mathrm{M} := s}] & \triangleq & x \\ \mathrm{M}_i[t_1, \ldots, t_{m_i}]\,[\overrightarrow{\mathrm{M} := s}] & \triangleq & s_i\,[x_1 \mapsto t_1\,[\overrightarrow{\mathrm{M} := s}], \ldots, x_{m_i} \mapsto t_{m_i}\,[\overrightarrow{\mathrm{M} := s}]] \\ f(\overrightarrow{y_1}.s_1, \ldots, \overrightarrow{y_m}.s_m)\,[\overrightarrow{\mathrm{M} := s}] & \triangleq & f(\overrightarrow{y_1}.s_1\,[\overrightarrow{\mathrm{M} := s}], \ldots, \overrightarrow{y_m}.s_m\,[\overrightarrow{\mathrm{M} := s}]) \end{array}$$

where $[\overrightarrow{\mathrm{M} := s}]$ denotes $[\mathrm{M}_1 := s_1, \ldots, \mathrm{M}_k := s_k]$.

Figure 6 Substitution for metavariables.

actually defines a $(\Sigma_c, \mathcal{E})$-structure by $[\![f]\!]^M_A(t_1, \ldots, t_n) \triangleq (\!|f|\!)^M \circ \langle t_1, \ldots, t_n \rangle$ for any A in $\mathcal{C}$. We say that an iteration functor $F : \mathcal{C} \to \mathcal{D}$ *preserves c-structures* if $F([\![c]\!]^M) = [\![c]\!]^N$, and $F((\!|f|\!)^M) = (\!|f|\!)^N$ for every $f \in \Sigma_c$.

▶ **Example 3.4** (The cyclic list datatype `CList`)**.** A `CList`-structure is given by a $(\Sigma_{\mathtt{CList}}, \mathsf{AxCy})$-structure M having the interpretations of "$[\,]$" and "::".

▶ **Example 3.5** (The cyclic tree type `CTree`)**.** A `CTree`-structure is given by a $(\Sigma_{\mathtt{CTree}}, \mathsf{AxBr} \cup \mathsf{AxBr}([\,], +))$-structure M where $[\![\mathtt{CTree}]\!]^M = N$ and N is a commutative monoid object $(N, \eta : \mathbf{1} \to N,\ \mu : N \times N \to N)$ in $\mathcal{C}$ satisfying $\mu^\dagger = \mathrm{id}_N$, where $(\!|\,[\,]\,|\!)^M = \eta$, $(\!|+|\!)^M = \mu$. It satisfies $\mathsf{AxBr}([\,], +)$. Note that any `CTree`-structure is always a *degenerated commutative bialgebra* (cf. [16]) in a cartesian category $\mathcal{C}$, i.e. N is also a comonoid $(N, !, \Delta)$ that satisfies the compatibility

$$\Delta \circ \eta = \eta \times \eta, \quad \Delta \circ \mu \ = \ (\mu \times \mu) \circ (\mathrm{id} \times \langle \pi_2, \pi_1 \rangle \times \mathrm{id}) \diamond (\Delta \times \Delta), \quad \mu \circ \Delta \ = \ \mathrm{id}.$$

The last equation is by $\mu \circ \Delta = \mu \circ \langle \mathrm{id}, \mathrm{id} \rangle = \mu \circ \langle \mathrm{id}, (\mu)^\dagger \rangle =^{(\mathrm{dinat})} (\mu)^\dagger = \mathrm{id}$. Thus, a `CTree`-structure models branch and sharing of cyclic sharing trees.

We next give a syntactic category and a Σ-structure to prove categorical completeness. Let Σ be a signature, and $\mathcal{E}$ a set of axioms which is the union of AxCy and axioms for all datatype declarations of base types c. Given axioms $\mathcal{E}$, all proved equations $\Gamma \vdash s = t \ : \overrightarrow{c}$ (which must be the empty metavariable context) by the second-order equational logic (Fig. 5), defines an equivalence relation $=_{\mathcal{E}}$ on well-typed terms, where we also identify renamed terms by bijective renaming of free and bound variables. We write an equivalence class of terms by $=_{\mathcal{E}}$ as $[\Gamma \vdash t \ : \overrightarrow{c}]_{\mathcal{E}}$. We define the category $\mathbf{Tm}(\mathcal{E})$ of terms by taking

- objects: sequences of base types $\overrightarrow{c}$
- morphisms: $[\Gamma \vdash t \ : \overrightarrow{c}]_{\mathcal{E}} : [\![\Gamma]\!] \to [\![\overrightarrow{c}]\!]$, the identity: $[\overrightarrow{x : c} \vdash \langle \overrightarrow{x} \rangle \ : \overrightarrow{c}]_{\mathcal{E}}$
- composition: $[\overrightarrow{x : b} \vdash s \ : \overrightarrow{c}]_{\mathcal{E}} \circ [\Gamma \vdash t \ : \overrightarrow{b}]_{\mathcal{E}} \triangleq [\Gamma \vdash (\overrightarrow{x}.s) \diamond t \ : \overrightarrow{c}]_{\mathcal{E}}$

▶ **Proposition 3.6.** *$\mathcal{Tm}(\mathcal{E})$ is an iteration category, and has a $(\Sigma, \mathcal{E})$-structure* U*.*

Proof. We define a Σ-structure U by $[\![c]\!]^{\mathrm{U}} \triangleq c$ for each $c \in \mathcal{B}$ and $[\![f]\!]^{\mathrm{U}}_{\overrightarrow{a}} \triangleq f$ for each function symbol f and arbitrary base types $\overrightarrow{a}$. We take

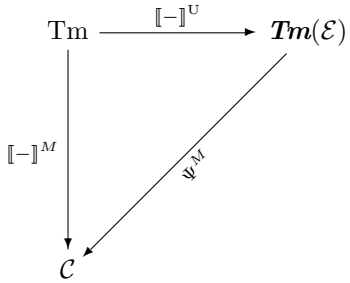
- terminal object: $()$ • pair: $\langle [s]_{\mathcal{E}}, [t]_{\mathcal{E}} \rangle \triangleq [\Gamma \vdash \langle s\,,\, t \rangle \ : \overrightarrow{c_1}, \overrightarrow{c_2}]_{\mathcal{E}}$
- product: concatenation of sequences
- Conway: $([\Gamma, \overrightarrow{x : c} \vdash t \ : \overrightarrow{c}]_{\mathcal{E}})^\dagger = [\Gamma \vdash \mathsf{cy}(\overrightarrow{x^c}.t) \ : \overrightarrow{c}]_{\mathcal{E}}$
- projections: $[x_1 : c_1, x_2 : c_2 \vdash x_i \ : c_i]_{\mathcal{E}}$

Then these data satisfy that $\mathbf{Tm}(\mathcal{E})$ is an iteration category and U forms a $(\Sigma, \mathcal{E})$-structure because of the axioms $\mathcal{E}$ for each $c \in \mathcal{B}$. Moreover, $([\![c]\!]^{\mathrm{U}}, (f)_{f \in \Sigma_c})$ is a c-structure. ◀

Then $[\![t]\!]^{\mathrm{U}} = [t]_{\mathcal{E}}$ holds for all well-typed terms t. Using it, we have the following.

▶ **Theorem 3.7** (Categorical soundness and completeness). *$\Gamma \vdash s = t \ : \overrightarrow{c}$ is derivable iff $[\![s]\!]^M_{\mathcal{C}} = [\![t]\!]^M_{\mathcal{C}}$ holds for all iteration categories $\mathcal{C}$ and all $(\Sigma, \mathcal{E})$-structures in $\mathcal{C}$.*

▶ **Theorem 3.8.** *For a $(\Sigma, \mathcal{E})$-structure M in an iteration category $\mathcal{C}$, there exists a unique iteration functor $\Psi^M : \boldsymbol{Tm}(\mathcal{E}) \longrightarrow \mathcal{C}$ that preserves $(\Sigma, \mathcal{E})$-structures. Pictorially, it is expressed as the following picture, where* Tm *denotes the set of all terms (without quotient).*



Proof. We write simply Ψ for Ψ^M. Since Ψ preserves $(\Sigma, \mathcal{E})$-structures, $\Psi([\![-]\!]^{\mathrm{U}}) = [\![-]\!]^M$ holds. Hence $\Psi([\![t]\!]^{\mathrm{U}}) = \Psi([t]_{\mathcal{E}}) = [\![t]\!]^M$ for any t, meaning that the mapping Ψ is required to satisfy

$$
\begin{array}{lll}
\Psi(\ [\Gamma \vdash y_i \ : c]_{\mathcal{E}} &) = \ \pi_i & \Psi([\Gamma \vdash \langle\,\rangle \ : ()]_{\mathcal{E}}) = ! \\
\Psi(\ [\Gamma \vdash \langle s\,,\, t\rangle \ : \overrightarrow{c_1}, \overrightarrow{c_2}]_{\mathcal{E}} &) = \langle \Psi[\Gamma \vdash s \ : \overrightarrow{c_1}]_{\mathcal{E}}, \Psi[\Gamma \vdash t \ : \overrightarrow{c_2}]_{\mathcal{E}} \rangle & \\
\Psi(\ [\Gamma \vdash \mathsf{cy}(\overrightarrow{x^c}.t) \ : \overrightarrow{c}]_{\mathcal{E}} &) = (\Psi[\Gamma, \overrightarrow{x : c} \vdash t \ : \overrightarrow{c}]_{\mathcal{E}})^{\dagger} & \\
\Psi(\ [\Gamma \vdash f(\ \overrightarrow{x_1^{a_1}}.t_1, \ldots, \overrightarrow{x_m^{a_m}}.t_m) \ : c]_{\mathcal{E}} &) = [\![f]\!]^M_{[\![\Gamma]\!]}(\Psi[\Gamma, \overrightarrow{x_1 : a_1} \vdash t_1 \ : b_1]_{\mathcal{E}}, \ldots) & \\
\Psi(\ [\Gamma \vdash (\overrightarrow{x^b}.t) \diamond s \ : c]_{\mathcal{E}} &) = \Psi[\Gamma, \overrightarrow{x : b}, \vdash t \ : c]_{\mathcal{E}} \circ \langle \mathrm{id}_{[\![\Gamma]\!]}, \Psi[\Gamma \vdash s \ : \overrightarrow{b}]_{\mathcal{E}} \rangle &
\end{array}
\tag{2}
$$

The above equations mean that Ψ is an iteration functor that sends the $(\Sigma, \mathcal{E})$-structure U to M. Such Ψ is uniquely determined by these equations because U is a $(\Sigma, \mathcal{E})$-structure. ◀

4 Fold on Cyclic Datatype

Fix a cyclic datatype c (say, the type `CList` of cyclic lists). By the previous theorem, for a c-structure M, the interpretation $[\![-]\!]^M$ determines a c-structure preserving iteration functor Ψ^M. If we take the target category $\mathcal{C}$ as also $\mathbf{Tm}(\mathcal{E})$, M should be another cyclic datatype b (say, the `CNat` of cyclic natural numbers), where the constructors of c are interpreted as terms of type b. For example, the sum of a cyclic list in Introduction is understood in this way. Thus the functor Ψ^M determined by $[\![-]\!]^M$ can be understood as a transformation of cyclic data from terms of type c to terms of type b.

Along this idea, we formulate the fold operation from the cyclic datatype c to b by the functor Ψ^M. Let (M, α) be an arbitrarily c-structure in $\mathbf{Tm}(\mathcal{E})$, where $[\![c]\!]^M = b \in \mathcal{B}$. We write the arrow part function Ψ^M on hom-sets as the fold, i.e.

$$
\mathbf{fold}^c_b(\alpha) : \mathbf{Tm}(\mathcal{E})([\![\Gamma]\!]^{\mathrm{U}},\ c) \longrightarrow \mathbf{Tm}(\mathcal{E})([\![\Gamma]\!]^M,\ b).
$$

4.1 Formalising fold as a second-order algebraic theory

The **fold** is a function on equivalence classes of term judgments modulo $\mathcal{E}$ including $\mathsf{AxCy} \cup \mathsf{AxBr}$ characterised by (2). Equivalently, we regard it as a function on terms (judgments) that preserves $=_{\mathcal{E}}$, i.e. $s =_{\mathcal{E}} t \;\Rightarrow\; \mathbf{fold}^c_b(\alpha)(s) =_{\mathcal{E}} \mathbf{fold}^c_b(\alpha)(t)$. In this subsection, we axiomatise the function $\mathbf{fold}^c_b$ as the laws of fold within second-order equational logic using (2).

Formalising a c-structure (M, α). To give $\alpha = (([\![f]\!])^M : [\![a_1]\!]\times\ldots\times[\![a_n]\!] \to [\![c]\!])_{f:a_1,\ldots,a_n\to c\in\Sigma_c}$ is to give terms $x_1 : [\![a_1]\!], \ldots, x_n : [\![a_n]\!] \vdash e_f \;\; : b$ for all $f : a_1, \ldots, a_n \to c \in \Sigma_c$ such that $([\![f]\!])^M = [e_f]_{\mathcal{E}}$. We represent α as a tuple of terms e_f according to function symbols in Σ_c by the order of datatype constructors listed in a `ctype` declaration of c.

Formalising fold. We next formalise the fold operation in second-order algebraic theory. The type of fold may be chosen as $\mathsf{fold}^c_b : (\overrightarrow{a_1} \to b), \ldots, (\overrightarrow{a_k} \to b), (c^m \to c) \to (b^m \to b)$, where the first k-arguments correspond to the c-structure α. But in second-order algebraic theory, the codomain of function symbol *must be* a sequence of base types (§2.1), so the current codomain $(b^m \to b)$ is inappropriate. To solve it, we introduce a new base type jty^b_m as the type of "encoded judgments" and a function symbol $\mathsf{judgmt}^b_m : (b^m \to b) \to \mathsf{jty}^b_m$ for each $m \in \mathbb{N}, b \in \mathcal{B}$. We encode a judgment $\overrightarrow{y : b} \vdash t \;\; : b$ as a term $\mathsf{judgmt}^b_m(\overrightarrow{y}.t)$, for $m = |\overrightarrow{y}|$, which will be denoted by $\overrightarrow{y} \Vdash t$ for readability. In case of $m = 0$, $\mathsf{jty}^b_0 = b$ and we do not use the constructor judgmt^b_0. In summary, the fold is formalised as the function symbol of the type

$$\mathsf{fold}^c_b : (\overrightarrow{a_1} \to b),\; \ldots, (\overrightarrow{a_k} \to b), \mathsf{jty}^c_m \to \mathsf{jty}^b_m$$

and the mathematical expression $\mathbf{fold}^c_b(\alpha)([\Gamma \vdash t \;\; : c]_{\mathcal{E}})$ at the level of semantics is formalised as a term $\mathsf{fold}^c_b(e_1, \ldots, e_k, \Gamma \Vdash t)$ in second-order algebraic theory, where each e_i corresponds to $([\![f_i]\!])^M$ for $f_i \in \Sigma_c$ in α.

Finally, we axiomatise fold by using the characterisation (2) in case of particular category $\mathcal{C} = \mathbf{Tm}(\mathcal{E})$ and a c-structure. Here we assume an additional function symbol $\mathsf{app} : (\overrightarrow{a} \to b), \overrightarrow{a} \to b$. We give the axioms FOLD in Fig. 7, which is straightforward formalisation of (2) in case of the target c-structure is given by terms of type b. The arguments of fold expressing the c-structure are abbreviated as E for simplicity. We also include the axioms and theorems (8)-(12) taken from AxCy and AxBr for simplification. This importation of several axioms from $\mathsf{AxCy} \cup \mathsf{AxBr}$ to the second-order algebraic theory FOLD is harmless because our general framework is second-order equational logic under $\mathcal{E} \cup \mathsf{FOLD}$ which includes $\mathsf{AxCy} \cup \mathsf{AxBr}$. The following is immediate by construction.

▶ **Proposition 4.1.** *Using the above formalisation process, the following are equivalent.*

- $\mathbf{fold}^c_b(\alpha)([\Gamma \vdash t \;\; : c]_{\mathcal{E}}) = [\Gamma' \vdash u \;\; : b]_{\mathcal{E}}$
- $\vdash \mathit{fold}(e_1, \ldots, e_k, \Gamma \Vdash t) = (\Gamma' \Vdash u) \;\; : \mathit{jty}^b_m$ *is derived from the axioms* $\mathcal{E} \cup \mathit{FOLD}$ *using the second-order equational logic.*

where α, e_i *and* t *are* fold *free,* $\Gamma = x_1 : c, \ldots, x_m : c$, $\Gamma' = x_1 : b, \ldots, x_m : b$.

▶ **Example 4.2.** The plus function on `CNat` can be defined as fold as follows.

```
plus : CNat,CNat → CNat
spec plus(m, n) = pl(m)
  where pl(0)    = n
        pl(S(m)) = S(pl(m))
fun plus(m, n) = fold (n, x.S(x)) m
```

Fold

(1) $\mathsf{fold}(E,\ \overrightarrow{y^c} \Vdash y_i) = \overrightarrow{y^b} \Vdash y_i$ (for $y_i \in \{\overrightarrow{y}\}$)
(2) $\mathsf{fold}(E,\ \overrightarrow{y} \Vdash \langle\,\rangle) = \overrightarrow{y} \Vdash \langle\,\rangle$
(3) $\mathsf{fold}(E,\ \overrightarrow{y} \Vdash \langle \mathrm{S}[\overrightarrow{y}], \mathrm{T}[\overrightarrow{y}]\rangle) = \overrightarrow{y} \Vdash \langle \mathsf{app}(\mathsf{fold}(E,\ \overrightarrow{y} \Vdash \mathrm{S}[\overrightarrow{y}]), \overrightarrow{y}),\ \mathsf{app}(\mathsf{fold}(E,\ \overrightarrow{y} \Vdash \mathrm{T}[\overrightarrow{y}]), \overrightarrow{y})\rangle$
(4) $\mathsf{fold}(E,\ \overrightarrow{y} \Vdash \mathsf{cy}(\overrightarrow{x}.\mathrm{T}[\overrightarrow{y}, \overrightarrow{x}])) = \overrightarrow{y} \Vdash \mathsf{cy}(\overrightarrow{x}.\mathsf{app}(\mathsf{fold}(E,\ \overrightarrow{y}, \overrightarrow{x} \Vdash \mathrm{T}[\overrightarrow{y}, \overrightarrow{x}]), \overrightarrow{y}, \overrightarrow{x}))$
(5) $\mathsf{fold}(E,\ \overrightarrow{y} \Vdash d(\overrightarrow{\mathrm{A}}, \mathrm{T}_1[\overrightarrow{y}], \ldots, \mathrm{T}_n[\overrightarrow{y}])) = \overrightarrow{y} \Vdash (\overrightarrow{x}.\mathrm{E}_d[\overrightarrow{\mathrm{A}}, \overrightarrow{x}]) \diamond \langle \mathsf{app}(\mathsf{fold}(E,\ \overrightarrow{y} \Vdash \mathrm{T}_1[\overrightarrow{y}]), \overrightarrow{y}), \ldots, \rangle$
(6) $\mathsf{fold}(E,\ x \Vdash (\overrightarrow{y}.\mathrm{T}[\overrightarrow{y}]) \diamond \mathrm{S}[\overrightarrow{x}]) = \overrightarrow{x} \Vdash \overrightarrow{y}.\mathsf{app}(\mathsf{fold}(E,\ \overrightarrow{y} \Vdash \mathrm{T}[\overrightarrow{y}]), \overrightarrow{y}) \diamond \mathsf{app}(\mathsf{fold}(E,\ \overrightarrow{x} \Vdash \mathrm{S}[\overrightarrow{x}]), \overrightarrow{x})$
(7) $\mathsf{app}(\overrightarrow{x} \Vdash \mathrm{S}[\overrightarrow{x}], z_1, \ldots, z_m) = \mathrm{S}[z_1, \ldots, z_m]$

Bekič and cycle cleaning

(8) $\mathsf{cy}(\overrightarrow{x}, \overrightarrow{y}.\langle \widehat{\mathrm{T}}, \widehat{\mathrm{S}}\rangle) = \langle \mathsf{cy}(\overrightarrow{x}.(\overrightarrow{y}.\widehat{\mathrm{T}}) \diamond \mathsf{cy}(\overrightarrow{y}.\widehat{\mathrm{S}})),\ \mathsf{cy}(\overrightarrow{y}.(\overrightarrow{x}.\widehat{\mathrm{S}}) \diamond \mathsf{cy}(\overrightarrow{x}.(\overrightarrow{y}.\widehat{\mathrm{T}}) \diamond \mathsf{cy}(\overrightarrow{y}.\widehat{\mathrm{S}})))\rangle$
(9) $\mathsf{cy}(\overrightarrow{y}.\mathrm{T}) = \mathrm{T}$ (NB. T cannot contain y)
(10) $\mathsf{cy}(x^c.x) = [\,]\quad \mathsf{cy}(x^c.\mathrm{T} + x) = \mathrm{T}$ (if a type c has $[\,]$ and "+" satisfying AxBr)

Composition

(11) $(\overrightarrow{y}.\mathrm{T}[\overrightarrow{y}]) \diamond \langle \mathrm{S}_1, \ldots, \mathrm{S}_n\rangle = \mathrm{T}[\mathrm{S}_1, \ldots, \mathrm{S}_n]$

Here E is a sequence $(\mathrm{E}_d)_{d\in\Sigma_c}$ of metavariables and $d \in \Sigma_c$.
In (8), $\widehat{\mathrm{T}}$ and $\widehat{\mathrm{S}}$ are short for $\mathrm{T}[\overrightarrow{x}, \overrightarrow{y}]$ and $\mathrm{S}[\overrightarrow{x}, \overrightarrow{y}]$, respectively.

Figure 7 Second-order algebraic theory FOLD of fold from the datatype c to b.

In specification, we understand `plus` in terms of a unary function `pl` which recurses on the first argument m and gives the second argument n if $m = 0$. Hence it is fold where the parameter n is passed to the Σ-structure of fold.

4.2 Primitive recursion by fold

The fold formalised above covers the ordinary fold on algebraic datatypes. Thus, we expect that various techniques on fold developed in functional programming, such as the fold fusion technique and representation of recursion principles such as [28] may be transferred to the current setting. Here we consider a way to implement a particular pattern of recursion appearing often in specifications as a fold. Consider a specification having a clause

$$\texttt{spec}\ f(d(t)) = e$$

where e contains $f(t)$ as well as t solely (cf. examples in §6). (If e constrains only the recursive call $f(t)$, it is merely a pattern of structural recursion, so it can be implemented by fold using the structure $x.e'$ where all the recursive calls $f(t)$ in e are abstracted to x as Example 4.2.)

The above specification (i.e. the clause with `spec` keyword) can be seen as describing primitive recursion, because it is similar the primitive recursion on natural numbers $f(S(n)) = e(f(n), n)$, where both n and $f(n)$ can be used at the right-hand side. In functional programming, it is known that primitive recursion on algebraic datatypes can be represented as fold, called paramorphism [27]. We sketch how we can import this technique (see also [21, §3.5], [22, §4.2]). For the above case, we take the fold where the target Σ-structure is the product b, b of types, i.e. $\mathsf{fold}^c_{b,b}$. In this case, variables in context are doubled at the right-hand sides of the axioms FOLD, e.g. for (1) $\mathsf{fold}(E,\ \overrightarrow{y} \Vdash y_i) = (\overrightarrow{y}, \overrightarrow{y'} \Vdash \langle y_i, y'_i\rangle)$. Let $\pi_1 = (x, y.x)$. We implement f as

$$f(\Gamma \Vdash t) \triangleq \pi_1 \diamond \mathsf{fold}(\cdots,\ \langle x, y.e',\ d(y)\rangle,\ \cdots,\ \Gamma \Vdash t)$$

where e' is obtained from e in the specification by replacing every "$f(t)$" with x and every "t" not in the form $f(t)$ with y. The other components of the c-structure for fold are implemented

Fold

(1) $\mathsf{fold}(E,\ \overrightarrow{y^{\mathsf{Var}_c}}.\,\mathsf{v}(y_i)) \to \overrightarrow{y^{\mathsf{Var}_b}}.\,\mathsf{v}(y_i)$

(2) $\mathsf{fold}(E,\ \overrightarrow{y}.\,\langle\,\rangle) \to \overrightarrow{y}.\,\langle\,\rangle$

(3) $\mathsf{fold}(E,\ \overrightarrow{y}.\,\langle \mathrm{S}[\overrightarrow{y}], \mathrm{T}[\overrightarrow{y}]\rangle) \to \overrightarrow{y}.\,\langle \mathsf{fold}(E,\ \overrightarrow{y}.\,\mathrm{S}[\overrightarrow{y}])@\overrightarrow{y},\ \mathsf{fold}(E,\ \overrightarrow{y}.\,\mathrm{T}[\overrightarrow{y}])@\overrightarrow{y}\rangle$

(4) $\mathsf{fold}(E,\ \overrightarrow{y}.\,\mathsf{cy}^1(x.\mathrm{T}[\overrightarrow{y}, x])) \to \overrightarrow{y}.\,\mathsf{cy}^1(x.\mathsf{fold}(E,\ \overrightarrow{y}, x.\,\mathrm{T}[\overrightarrow{y}, x])@\overrightarrow{y}, x)$

(5) $\mathsf{fold}(E,\ \overrightarrow{y}.\,d(\overrightarrow{\mathrm{A}}, \mathrm{T}_1[\overrightarrow{y}], \ldots, \mathrm{T}_n[\overrightarrow{y}])) \to \overrightarrow{y}.\,(\overrightarrow{x}.\mathrm{E}_d[\overrightarrow{\mathrm{A}}, \overrightarrow{x}]) \diamond \langle\, \mathsf{fold}(E,\ \overrightarrow{y}.\,\mathrm{T}_1[\overrightarrow{y}])@\overrightarrow{y}, \ldots\rangle$

(6) $\mathsf{fold}(E,\ \overrightarrow{x}.\,(\overrightarrow{y}.\mathrm{T}[\overrightarrow{y}]) \diamond \mathrm{S}[\overrightarrow{x}]) \to \overrightarrow{x}.\,(\overrightarrow{y}.\mathsf{fold}(E,\ \overrightarrow{y}.\,\mathrm{T}[\overrightarrow{y}])@\overrightarrow{y})\ \diamond\ \mathsf{fold}(E,\ \overrightarrow{x}.\,\mathrm{S}[\overrightarrow{x}])@\overrightarrow{x}$

Bekič and cycle cleaning (for $m, n \geq 1$)

(8) $\mathsf{cy}^{m+n}(\overrightarrow{x}, \overrightarrow{y}.\,\langle \widehat{\mathrm{T}},\ \widehat{\mathrm{S}}\rangle) \to \langle\, \mathsf{cy}^m(\overrightarrow{x}.\,(\overrightarrow{y}.\widehat{\mathrm{T}}) \diamond \mathsf{cy}^n(\overrightarrow{y}.\widehat{\mathrm{S}})),\ \mathsf{cy}^n(\overrightarrow{y}.\,(\overrightarrow{x}.\widehat{\mathrm{S}}) \diamond \mathsf{cy}^m(\overrightarrow{x}.\,(\overrightarrow{y}.\widehat{\mathrm{T}}) \diamond \mathsf{cy}^n(\overrightarrow{y}.\widehat{\mathrm{S}})))\rangle$

(9) $\mathsf{cy}(\overrightarrow{y}.\mathrm{T}) \to \mathrm{T}$

(10) $\mathsf{cy}(x.\mathsf{v}(x)) \to [\,]\quad \mathsf{cy}(x.\mathrm{T} + \mathsf{v}(x)) \to \mathrm{T}$ (if a type c has $[\,]$ and "+" satisfying AxBr)

Composition

(11) $(\overrightarrow{y}.\,\mathsf{v}(y_i)) \diamond \langle \overrightarrow{\mathrm{S}} \rangle \to \mathrm{S}_i$

(12) $(\overrightarrow{y}.\,d(\overrightarrow{x_1}.\,\mathrm{T}_1[\overrightarrow{y}, \overrightarrow{x_1}], \ldots)) \diamond \langle \overrightarrow{\mathrm{S}} \rangle \to d(\overrightarrow{x}.\,(\overrightarrow{y}.\,\mathrm{T}_1[\overrightarrow{y}, \overrightarrow{x_1}]) \diamond \langle \overrightarrow{\mathrm{S}} \rangle, \ldots, (\overrightarrow{y}.\,\mathrm{T}_n[\overrightarrow{y}, \overrightarrow{x_n}]) \diamond \langle \overrightarrow{\mathrm{S}} \rangle)$ (for each constructor d)

Figure 8 Rewrite system FOLDr.

by the same way, according to the specification. Then by induction on the structure of terms t, we have $\mathsf{fold}^c_{b,b}(E, \overrightarrow{y} \Vdash t) = \overrightarrow{y} \Vdash \langle\, \mathsf{app}(f(\overrightarrow{y} \Vdash t), \overrightarrow{y}),\ t\,\rangle$ for closed t using FOLD. By the characterisation (2), we have $f(d(t)) = (x, y.\,e') \diamond \langle f(t), t\rangle = e$, thus it satisfies the specification. We use extensively this technique in §6.

5 Strongly Normalising Computation Rules for FOLD

We expect that FOLD provides strong normalising computation rules. An immediate idea is to regard the axioms FOLD as rewrite rules by orienting each axiom from left to right.

But proving strong normalisation (SN) of FOLD is not straightforward. The sizes of both sides of equations in FOLD are not decreasing in most axioms. So, assigning some "measure" to the rules in FOLDr that is strictly decreasing is difficult for this case. If the axioms (regarded as rewrite rules) are a binding CRS [18] (meaning that every meta-application $\mathrm{M}[t_1, \ldots, t_n]$ is of the form $\mathrm{M}[\overrightarrow{x}]$), then it is possible to use a simple polynomial interpretation to prove termination of second-order rules [18]. Unfortunately, this is not the case because in (5) and (11) there are meta-applications violating the condition. Existence of meta-application means that it essentially involves the β-reduction, thus it has the same difficulty as proving strong normalisation of the simply-typed λ-calculus.

We use a general established method of *the General Schema*[4, 3], which is based on Tait's computability method to show SN. The General Schema has succeeded to prove SN of various recursors such as the recursor in Gödel's System T. The basic idea of the General Schema is to check whether the arguments of recursive calls in the the right-hand side of a rewrite rule are "smaller" than the left-hand sides' ones. It is similar to Coquand's notion of "structurally smaller" [8], but more relaxed and extended.

Rewrite rules using strictly positive types. In order to apply the General Schema criterion, we refine the second-order algebraic theory FOLD to the rewrite rule FOLDr. The General Schema in [3] is formulated for a framework of rewrite rules called inductive datatype systems, whose (essentially) second-order fragment is almost the same as the present formulation

given in §2. Minor differences are as follows.

(i) The target of function symbols must be a single (not necessary base) type in inductive datatype systems. Hence we introduce the product type constructor $\times$, assume that $b_1 \times b_2$ is again a base type in the sense of §2.2, and use it for the target type.

(ii) Instead of term $x_1, \ldots, x_n.t$ that binds a sequence of variables and is of sort $a_1, \ldots, a_n \to b$ in second-order algebraic theory, we use $x_1.\cdots.x_n.t$ that repeatedly binds single variables and is of type $a_1 \to \cdots \to a_n \to b$. Now the abbreviation $\overrightarrow{x}.t$ denotes $x_1.\cdots.x_n.t$.

(iii) We assume that a function symbol $@ : (a \to b), a \to b$ and a rule $(x.\mathrm{T}[x])@\mathrm{S} \to \mathrm{T}[\mathrm{S}]$ for application ([3] Def. 2, β-IDTS). We write $(\overrightarrow{x}.t)@\overrightarrow{s}$ for $(\overrightarrow{x}.t)@s_1 \cdots @s_n$.

The General Schema requires a notion of strictly positivity. Crucially, the constructors used in FOLD are *not strictly positive*, as cy and $\diamond$ involve negative occurrence of c in $(c \to c)$. We can overcome this problem by modifying the type $(c \to c)$ to a restricted $(\mathsf{Var}_c \to c)$, where Var_c is a base type having no constructor considered as a type of "variables" of type c. We assume the constructor v which embeds a "variable" into a term. We modify the types of constructors as follows:

$$\begin{array}{llll} \langle -, \cdots, - \rangle & : c_1, \ldots, c_n \to \overrightarrow{c}, & \mathsf{cy} & : (\overrightarrow{\mathsf{Var}_c} \to \overrightarrow{c}) \to \overrightarrow{c}, \\ \mathsf{v} & : \mathsf{Var}_c \to c, & - \diamond - & : (\overrightarrow{\mathsf{Var}_a} \to \overrightarrow{c}), a_1 \times \cdots \times a_n \to \overrightarrow{c}, \end{array}$$

where $\overrightarrow{c}$ denotes $c_1 \times \cdots \times c_n$, c_i's and a are base types, $\overrightarrow{\mathsf{Var}_a} \to \tau$ is short for $\mathsf{Var}_{a_1} \to \cdots \to \mathsf{Var}_{a_n} \to \tau$ and similarly for $\overrightarrow{\mathsf{Var}_c} \to \tau$. The use of a type $\mathsf{Var}_\sigma \to \tau$ to represent binders is known in the field of mechanized reasoning, sometimes called *(weak) higher-order abstract syntax* [9]. Accordingly, the type of fold is now

$$\mathsf{fold} : (\overrightarrow{\mathsf{Var}_{a_1}} \to b), \ldots, (\overrightarrow{\mathsf{Var}_{a_k}} \to b), (\mathsf{Var}_c^m \to c) \to (\mathsf{Var}_b^m \to b),$$

and rules are modified to FOLDr giving in Fig. 8. In case of inductive data type system, a term of the form $\overrightarrow{y}.t$ is allowed and well-typed (although not allowed as a sole term in case of second-order algebraic theory), thus we can now write the binder $\overrightarrow{y}.-$ directly at the right-hand side. FOLDr is correct.

▶ **Lemma 5.1.** *If $t \to^+_{\mathsf{FOLDr}} t'$ where t' does not involve @, then $\check{t} = \check{t'}$ is derivable from FOLD without using (Sym) where $\check{t}, \check{t'}$ recovers the original term notation from the encoding we gave above.*

▶ **Theorem 5.2.** *The rewrite system FOLDr is strongly normalising.*

Proof. Since FOLDr fits into the General Schema using the well-founded order

$$\mathsf{fold} > \mathsf{cy}^m > \mathsf{cy}^n > \diamond > \mathsf{v} > \text{any other constructors}, @$$

for natural numbers $m > n$, it is strongly normalising. Note that the superscript of cy in (8) indicates the number of arguments (cf. §2.1). This kind of indication of an "invariant" is similar to the idea of higher-order semantic labelling [19], but here we just make the existing superscript explicit rather than labelling. ◀

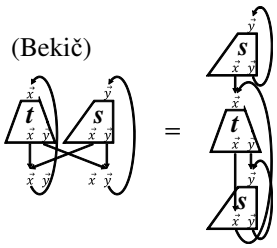
6 Computing by Fold on Cyclic Datatypes

In this section, we demonstrate fold computation on cyclic data by several examples.

▶ **Example 6.1.** As an example of primitive recursion on cyclic datatypes mentioned in §4.2. we consider the tail of a cyclic list, which we call `ctail`. It should satisfy the specification

below right. But how to define the tail of cy-term is not immediately clear. For example, what should be the result of `ctail (cy(x.1::2::x))`? This case may need unfolding of cycle as in [17]. A naive unfolding by using the fixed point law $\mathsf{cy}(x.t) = t\,[\mathsf{cy}(x.t)/x]$ violates strong normalisation because it copies the original term. It actually *increases* complexity.

```
ctail : CList → CNat
spec ctail ([])      = []
     ctail (k::t)    = t
     ctail (cy(x. t))= ??
```



Rather than the fixed point law, we use another important principle of cyclic structures known as *Bekič law*, given by the axiom (Bekič) in AxCy or (8) in FOLDr. It says that the fixed point of a pair can be obtained by computing the fixed point of each of its components independently and composing them suitably (see the right figure). It can be seen as *decreasing* complexity of cyclic computation because looking at the argument of cy, the number of components of tuple is reduced. We define `ctail` by fold.

```
fun ctail(t) = π1 ◇ fold (<[],[]>, k.x.y.<y, k::y>) t
```

```
ctail(cy(x.1::2::x)) →+ π1 ◇ cy(x.y. <2::y, 1::2::y>)
    →+ π1 ◇ <cy(x.2::cy(y.1::2::y)), cy(y.1::2::y)>
     →  cy(x.2::cy(y.1::2::y)) → 2::cy(y.1::2::y)      (Normal form)
```

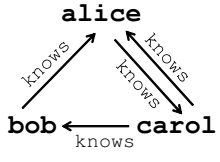
Note that the above normal form does not mean a head normal form and we do not rely on lazy evaluation. The highlighted step uses Bekič law.

▶ **Example 6.2.** This example shows that our cyclic datatype has ability to express directed graphs. The graph shown below right represents friend relationship, which describes Alice knows Carol, Bob knows Alice, and Carol knows Alice and Bob. This is represented as a term

```
cy(a.b.c.<name("alice")+knows(c), name("bob")+knows(a),
          name("carol")+knows(a)+knows(b)>)
```

which we call `g`. The term `g` is of type `FriendGraph` defined as follows.

```
ctype FriendGraph where
  knows : FriendGraph → FriendGraph
  name  : String → FriendGraph
  [] : FriendGraph
  + : FriendGraph,FriendGraph → FriendGraph
with axioms AxCy,AxBr([],+)
```



We define a function `collect` that collects all names in a graph as a name list of type `Names`.

```
ctype Names where
  nm : String → Names
  [] : Names
  + : Names,Names → Names
with axioms AxCy,AxBr([],+)
```

```
collect : FriendGraph → Names
spec collect (knows(t)) = collect(t)
     collect (name(p))  = nm(p)

fun collect t = π1◇ folde t
```

Then we collect certainly all names by FOLDr as follows, where `folde` is short for `fold (x.y.<x,knows(y)>, x.y.<nm(y),name(y)>)`.

```
collect g = π1◇ folde g
→ π1◇ (cy(a.a'.b.b'.c.c'.
    <folde(a.b.c.name("alice")+knows(c)), folde(a.b.c.name("bob")+knows(a)),
     folde(a.b.c.name("carol")+knows(a)+knows(b)>)))
→+ π1◇ (cy(a.a'.b.b'.c.c'.
  < <nm("alice"),name("alice")>, <nm("bob"),name("bob")>, <nm("carol"),name("carol")> >)
→+ nm("alice")+nm("bob")+nm("carol")
```

7 Related Work

There has been various work to deal with graph computation and cyclic data structures in functional programming and foundational calculi including [12, 30, 6, 24, 26, 2, 1]. Several work [12, 30, 26] relies on lazy evaluation to deal with cycles. The present paper is different in this respect. We do not assume any particular operational semantics nor strategy to obtain strongly normalising fold on cyclic data. This point may be useful to deal with cyclic datatypes in proof assistance like Coq or Agda.

Foundational graph rewriting calculi, such as equational term graph rewriting systems [2], are general frameworks of graph computation. The fold on cyclic datatype in this paper is more restricted than general graph rewriting. However, our emphasis is clarification of the categorical and algebraic structure of cyclic datatypes and the computation fold on them by regarding fold as a structure preserving map, rather than unrestricted rewriting. It was a key to obtain strong normalisation. We also hope that it will be useful for further optimisation such as the fold fusion based on semantics as done in [22] Sec. 4.3. The general study of graph rewriting was also important for our study at the foundational level. The unit "[]" of branching in AxBr corresponds to the black hole constant "•" considered in [2], due to [5]. This observation has been used to give an effective operational semantics of graph transformation in [26].

In [17, 20], the present author aimed to capture the unique representations of cyclic sharing data structures (without any quotient) in order to obtain efficient functional programming concept. The approach taken in this paper is different. We have assumed the axioms AxCy and AxBr to equate bisimilar graphs. The point is that bisimulation on graphs can be efficiently decidable [10], thus now we regard that uniqueness of representation is not quite serious.

In [21, 22], the author and collaborators gave algebraic and categorical semantics of a graph transformation language UnCAL [6, 24] using iteration categories [5]. The graph data of UnCAL corresponds to cyclic sharing trees of type `CTree` in the present paper. UnCAL does not have the notion of types. Hence structural recursive functions in UnCAL are always transformations from general graphs to graphs, thus typing such as `sum:CList→CNat` (in Introduction) or `collect:FriendGraph→Names` (in §6) could not be formulated. The present paper advanced one step further by developing a suitable algebraic framework that captures datatypes supporting cycles and sharing. We have used a rewriting technique of

the General Schema [3] to show strong normalisation (not merely termination of a particular computation strategy or algorithm) of fold. Such a direction has not been pursued so far.

Acknowledgments. I am grateful to Kazutaka Matsuda and Kazuyuki Asada for various discussions about calculi and programming languages about graphs.

References

1. Z. M. Ariola and S. Blom. Cyclic lambda calculi. In *Theoretical Aspects of Computer Software, LNCS 1281*, pages 77–106, 1997.
2. Z. M. Ariola and J. W. Klop. Equational term graph rewriting. *Fundam. Inform.*, 26(3/4):207–240, 1996.
3. F. Blanqui. Termination and confluence of higher-order rewrite systems. In *Rewriting Techniques and Application (RTA 2000)*, LNCS 1833, pages 47–61. Springer, 2000.
4. F. Blanqui, J.-P. Jouannaud, and M. Okada. Inductive data type systems. *Theoretical Computer Science*, 272:41–68, 2002.
5. S. L. Bloom and Z. Ésik. *Iteration Theories – The Equational Logic of Iterative Processes*. EATCS Monographs on Theoretical Computer Science. Springer, 1993.
6. P. Buneman, M. F. Fernandez, and D. Suciu. UnQL: A query language and algebra for semistructured data based on structural recursion. *VLDB J.*, 9(1):76–110, 2000.
7. C. Calcagno, P. Gardner, and U. Zarfaty. Context logic and tree update. In *Proc. of POPL'05*, pages 271–282, 2005. doi:10.1145/1040305.1040328.
8. T. Coquand. Pattern matching with dependent types. In *Proc. of the 3rd Work. on Types for Proofs and Programs*, 1992.
9. J. Despeyroux, A. Felty, and A. Hirschowitz. Higher-order abstract syntax in Coq. In *Typed Lambda Calculi and Applications, LNCS 902*, pages 124–138, 1995.
10. A. Dovier, C. Piazza, and A. Policriti. An efficient algorithm for computing bisimulation equivalence. *Theoretical Computer Science*, 311:221–256, 2004. Issues 1-3.
11. Z. Ésik. Axiomatizing iteration categories. *Acta Cybernetica*, 14:65–82, 1999.
12. L. Fegaras and T. Sheard. Revisiting catamorphisms over datatypes with embedded functions (or, programs from outer space). In *POPL'96*, pages 284–294, 1996. doi:10.1145/237721.237792.
13. M. Fiore and C.-K. Hur. Second-order equational logic. In *Proc. of CSL'10*, LNCS 6247, pages 320–335, 2010.
14. M. Fiore and O. Mahmoud. Second-order algebraic theories. In *Proc. of MFCS'10*, LNCS 6281, pages 368–380, 2010.
15. M. Fiore and S. Staton. Substitution, jumps, and algebraic effects. In *Proc. of LICS'14*, 2014.
16. M. P. Fiore and M. D. Campos. The algebra of directed acyclic graphs. In *Computation, Logic, Games, and Quantum Foundations*, LNCS 7860, pages 37–51, 2013.
17. N. Ghani, M. Hamana, T. Uustalu, and V. Vene. Representing cyclic structures as nested datatypes. In *Proc. of TFP'06*, pages 173–188, 2006.
18. M. Hamana. Universal algebra for termination of higher-order rewriting. In *Proc. of RTA'05*, LNCS 3467, pages 135–149, 2005.
19. M. Hamana. Higher-order semantic labelling for inductive datatype systems. In *Proc. of PPDP'07*, pages 97–108. ACM Press, 2007. doi:10.1145/1273920.1273933.
20. M. Hamana. Initial algebra semantics for cyclic sharing tree structures. *Logical Methods in Computer Science*, 6(3), 2010.

21 M. Hamana. Iteration algebras for UnQL graphs and completeness for bisimulation. In *Proc. of Fixed Points in Computer Science (FICS'15)*, Electronic Proceedings in Theoretical Computer Science 191, pages 75–89, 2015.

22 M. Hamana, K. Matsuda, and K. Asada. The algebra of recursive graph transformation language UnCAL: Complete axiomatisation and iteration categorical semantics. submitted.

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

24 S. Hidaka, Z. Hu, K. Inaba, H. Kato, K. Matsuda, and K. Nakano. Bidirectionalizing graph transformations. In *Proc. of ICFP 2010*, pages 205–216, 2010.

25 A. Joyal, R. Street, and D. Verity. Traced monoidal categories. *Mathematical Proceedings of the Cambridge Philosophical Society*, 119(3):447–468, 1996.

26 K. Matsuda and K. Asada. Graph transformation as graph reduction: A functional reformulation of graph-transformation language UnCAL. Technical Report GRACE-TR 2015-01, National Institute of Informatics, January 2015.

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

28 E. Meijer, M. M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Proc of FPCA'91*, pages 124–144, 1991.

29 R. Milner. A complete inference system for a class of regular behaviours. *J. Comput. Syst. Sci.*, 28(3):439–466, 1984.

30 B. C.d.S. Oliveira and W. R. Cook. Functional programming with structured graphs. In *Proc. of ICFP'12*, pages 77–88, 2012.

31 A. K. Simpson and G. D. Plotkin. Complete axioms for categorical fixed-point operators. In *Proc. of LICS'00*, pages 30–41, 2000.

32 S. Staton. An algebraic presentation of predicate logic. In *Proc. of FOSSACS 201*, pages 401–417, 2013.

33 S. Staton. Algebraic effects, linearity, and quantum programming languages. In *Proc. of POPL'15*, pages 395–406, 2015.

34 G. Winskel. *The Formal Semantics of Programming Languages*. The MIT Press, 1993.