Logical Methods in Computer Science
Vol. 13(4:8)2017, pp. 1–38
www.lmcs-online.org

Submitted Jul. 27, 2016
Published Nov. 15, 2017

CYCLIC DATATYPES MODULO BISIMULATION BASED ON SECOND-ORDER ALGEBRAIC THEORIES

MAKOTO HAMANA

Department of Computer Science, Gunma University
e-mail address: 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 Ésik. Thereby, the rules are correct by construction. 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. We also prove the property of "Church-Rosser modulo bisimulation" for the computation rules. Combining these results, we have a remarkable decidability result of the equational theory of cyclic data and fold.

1. Introduction

Cyclic data structures in functional programming are tricky to handle. In Haskell, one can define a cyclic data structure, such as cyclic lists by

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

The feasibility of such a recursive definition of cyclic data depends on lazy evaluation. For example, one can safely take the head of the cyclic list:

```
head clist   ⇝   2
```

However, this encoding is *not completely safe*. For example, consider the sum of all elements using the above recursive encoding. of `clist` in Haskell. It falls into non-termination:

```
sum clist   ⇝   non-termination
```

This means that such a naive encoding of cyclic structure does not ensure safety.

The computation using our framework is guaranteed to be *safe* meaning that it is always terminating, i.e., *strongly normalising*. 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 develop a framework for syntax and semantics of *cyclic datatypes* that makes this understanding and computation correct.

Key words and phrases: cyclic data structures, traced cartesian category, iteration theory, fixed point, categorical semantics, the General Schema, functional programming, fold.

DOI:10.23638/LMCS-13(4:8)2017

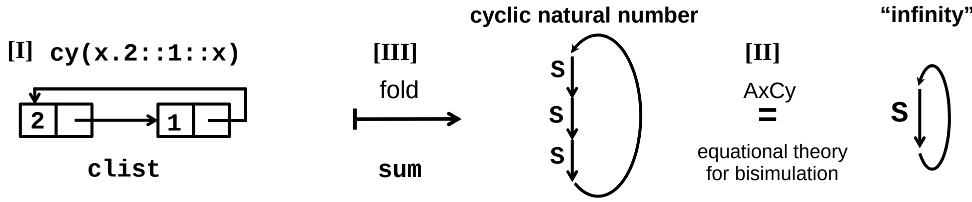


Figure 1: Framework: Second-order algebraic theories and iteration theories

Our framework of cyclic datatypes is founded on second-order algebraic theories of Fiore et al. [FH10, FM10]. Second-order algebraic theories are founded on the mathematical theory of second-order abstract syntax by Fiore, Plotkin, and Turi [FPT99, Ham04, Fio08] and have been shown to be a useful framework that models various important notions of programming languages, such as logic programming [Sta13], algebraic effects [FS14], quantum computation [Sta15]. This paper gives another application of second-order algebraic theories, namely, to cyclic datatypes and their computation. We use second-order algebraic theories to give a uniform setting for typed syntax, equational 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 [BÉ93]. Thereby the rules are correct by construction. Finally, we prove strong normalisation by using the General Schema criterion [Bla00] for second-order computation rules.

1.1. **Overview of computation.** 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, while intuitively understandable without going into the detailed theory.

First we consider an example of cyclic lists. The codes below with the keyword `ctype` are intended to declare cyclic datatypes. Here we declare the type `CNat` of natural numbers and the type `CList` of cyclic lists having ordinary constructors in Haskell or Agda style.

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

```
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. This is a fundamental idea presented in [FS96] [GHUV06].

A variable occurrence `x` in the body refers to the top, hence it makes a cycle. The terms built from the constructors of `CList` and the default constructor `cy` are required to satisfy the axioms AxCy (given later in Fig. 4) indicated by the keyword "`with axioms`" (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.

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

The above code with the keyword `spec` describes an equational *specification* of a function. It requires that the sum function from cyclic lists to cyclic natural numbers must satisfy the usual recursive properties of sum. We intend that the `spec` code is merely a (loose) specification, and not a definition, because it lacks the case of a `cy`-term. Here we assume that the `plus` function on `CNat` has already been defined (as presented later in Example 4.2). The following code with the keyword `fun` defines the function `sum`.

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

It is defined by the `fold` combinator on the cyclic datatypes. The first two arguments `0` and `k.x.plus(k,x)` correspond to the right-hand sides of the specification of `sum`, where `k.x.` are variable binders (as in λ-terms). The `fold` is actually the fold on a cyclic datatype, which knows how to cope with `cy`-terms. The sum of a cyclic list can be computed as follows:

$$
\begin{array}{l}
\mathtt{sum(cy(x.S^2(0)::S(0)::x))} \\
\to^{+}\ \mathtt{cy(x.\ fold\ (0,\ k.x.plus(k,x))\ (x.S^2(0)::S(0)::x;x))} \\
\to^{+}\ \mathtt{cy(x.S(S(S(x))))}
\end{array}
$$

where we represent a usual natural number n by $\mathtt{S}^n\mathtt{(0)}$. The final term is a normal form that cannot be rewritten further. Therefore, we regard it as the computation result. The steps presented above are actual rewrite steps by the second-order rewrite rules FOLDr given later in Fig. 7.

1.2. **Equational logical framework for cyclic computation.** 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 computation results. We use second-order equational logic and the axioms AxCy to equate cyclic data formally (Fig. 1 [II]). It completely characterises the notion of bisimulation on cyclic data. We can formally prove that an equation on cyclic data, such as

$$\mathtt{cy(x.S(S(S(x))))} \;=\; \mathtt{cy(x.S(x))} \tag{1.1}$$

is derivable from the axioms AxCy in the second-order equational logic. Since AxCy characterises bisimulation, it means that the expression `cy(x.S(S(S(x))))` is bisimilar to `cy(x.S(x))`, which is a minimal representation of the result regarded as ∞ (infinity). In this paper, we do not develop an explicit algorithm to extract such a minimal result from the computation result, but it is noteworthy that this equational theory generated by AxCy is decidable [BÉ93]. Consequently, it is computationally reasonable. More practical examples on cyclic datatypes and computation will be given in §7.

1.3. **Proof by rewriting.** In §6, we will develop a decidable proof method for equations involving functions defined by fold. We give an algorithm to prove whether an equation, e.g.

$$\mathtt{sum(cy(x.2::1::x))} \;=\; \mathtt{plus(\ sum(cy(x.4::5::x)),\ cy(x.x)\ )} \tag{1.2}$$

holds or not under AxCy. The algorithm is first rewriting both sides of the equation to normal forms,

$$\begin{tikzcd}
\mathtt{sum(cy(x.2::1::x))} \arrow[dd, "*"'] & \mathtt{plus(\ sum(cy(x.4::5::x)),\ cy(x.x)\ )} \arrow[d, "*"] \\
 & \mathtt{plus(cy(x.S^9(x)),\ cy(x.x))} \arrow[d, "*"] \\
\mathtt{cy(x.S^3(x))} \arrow[r, phantom, "\sim"] & \mathtt{cy(x.cy(x.S^9(x)))}
\end{tikzcd}$$

then comparing the normal forms by the bisimulation $\sim$ on cyclic data. In this case, these are actually bisimilar, hence we conclude (1.2) holds.

Why is this methodology correct? In this paper, we give rigorous reasons and proofs validating the methodology in §6. An important fact is that even when an equation involving cyclic data (as the above example) the method is ensured to be decidable. The rewrite relation "$\to$" has two important properties: strong normalisation (§5) and Church-Rosser modulo bisimulation (§6), both of which are necessary to establish this proof method. We will prove these rewriting properties by employing advanced rewriting techniques known as the General Schema [Bla00, Bla16], and local uniform coherence [JKR83].

Note that one can also show that an equation on cyclic data, such as (1.2), holds using ordinary domain theoretic semantics of lazy functional programs. A remarkable fact is that using our framework, we can show it without using domains, complete partial orders, or complex proof methods. Our methodology is simpler and decidable, i.e., strongly normalising rewriting and comparison by bisimulation.

1.4. **Previous approaches: cyclic structures and functional programming.** Fegaras and Sheard [FS96] established initial algebra semantics of mixed variant recursive types, and showed that cyclic structures (without any quotient) can be encoded by variable binding in higher-order abstract syntax (HOAS). It was a starting point of succeeding works [GHUV06, Ham09, Ham10, OC12, Ham12] improving their HOAS encoding (which had a few drawbacks) of cyclic structures.

Our previous four papers [GHUV06, Ham09, Ham10, Ham12] improved and extended it to various directions. Especially, [GHUV06, Ham09, Ham10] aimed to capture unique representations of cyclic sharing data structures (without any quotient) in order to obtain an efficient functional programming concept. In the present paper, we will assume the axioms AxCy and AxBr (Fig. 4) to equate bisimilar graphs. Uniqueness of representations is desirable, but uniqueness *up to bisimilarity* chosen in this paper is also desirable, because checking bisimilarity is efficiently decidable [DPP04] and reflects the meaning of cyclicity.

Oliveira and Cook [OC12] also improved [FS96] by using the parametric HOAS encoding [Chl08] of variable binding for cycle constructs (without any quotient), instead of HOAS, and developed generic fold combinators on them for functional programming.

In all the works mentioned in this subsection, bisimilarity was not used for identifications of cyclic structures.

1.5. **Our recent work.** In [Ham15, HMA], the author and collaborators gave algebraic and categorical semantics of a graph transformation language UnCAL [BFS00, HHI$^+$10] using iteration theories [BÉ93]. The graph data of UnCAL corresponds to cyclic sharing trees of type `CTree` in the present paper, where graphs are treated modulo bisimulation. UnCAL does not have the notion of types, thus structural recursive functions in UnCAL are always transformations from general graphs to graphs, thus typing such as `sum:CList`$\to$`CNat` (in Introduction) or `collect:FriendGraph`$\to$`Names` (in §7) could not be formulated. The present paper develops a suitable algebraic framework that captures datatypes supporting cycles and sharing.

1.6. **Novelty of this paper.** Since cyclic data structures is potentially dangerous because of cyclicity as exemplified in the begining of this section, termination of fold (or iterator) on cyclic structures should be ensured. However, *none of the previous works* including [FS96, GHUV06, Ham09, Ham10, Ham12, OC12, BFS00, HHI$^+$10, MA15] *on cyclic structures have formally proved termination nor strong normalisation of fold*, and the preservation of suitable equivalence on cyclic structures by fold. Moreover, these works did not developed a *decidable proof method* to prove equations on cyclic data with fold, such as the equation (1.2). In contrast to it, we will prove and provide a decidable proof method by showing rewriting properties of FOLDr, i.e., strong normalisation and Church-Rosser modulo bisimulation, which are new results.

1.7. **Organisation.** The paper is organised as follows. We first introduce cartesian second-order algebraic theories, which give syntax and equational logic of cyclic datatypes in §2. We next give a categorical semantics of cyclic datatypes in §3. We then extract the fold function from the categorical semantics in §4. In §5, we extract second-order rewrite rules of fold and show strong normalisation. In §6, we prove Church-Rosser modulo bisimulation of the rewrite system of fold and then obtain decidability of the equational theory. In §7, we consider several examples of computing by fold on cyclic datatypes. In §9, we summarise the paper and discuss related work.

This paper is the fully reworked and extended version of the conference paper [Ham16]. Besides proofs of all results, the present paper establishes Church-Rosser modulo bisimilarity (Thm. 6.8) and whence decidability of the theory of cyclic data and fold (Cor. 6.10). In addition, there is now a precise description of how primitive recursive functions are defined in our framework (§4.4), and additional examples are provided (§7).

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 second-order equational logic in [FH10] and [Ham07]. Here "cartesian" means that the target sort of a function symbol is a sequence of base types and we allow unrestricted substitution for variables. 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 the cycle construct 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 $f : \overrightarrow{\tau} \to c$, which is regarded as a constructor of c. A *metavariable* is a variable of (at most) first-order type, declared as $\textsc{m} : \overrightarrow{a} \to b$ (written as small-caps letters $\textsc{t}, \textsc{s}, \textsc{m}, \ldots$). A *variable* of a base type is merely called variable (written usually $x, y, \ldots$, or sometimes written x^b when it is of type b). The raw syntax is given as follows.

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

The last form is called a *meta-application*, meaning that when we instantiate $\textsc{m} : \overrightarrow{a} \to b$ with a term s, free variables of s (which are of types $\overrightarrow{a}$) are replaced with (meta-)terms $t_1, \ldots, t_n$ (cf. Def. 2.1). We may write $x_1, \ldots, x_n.\, t$ for $x_1. \cdots .x_n.\, t$, and we assume ordinary α-equivalence for bound variables. 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 (N.B. not meta-terms). Meta-terms are used for formulating equational axioms, which are expected to be instantiated to terms.

A metavariable context Θ is a sequence of (metavariable:type)-pairs, and a context Γ is a sequence of (variable:base 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. 2. We omit often 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 $t\,\{x_1 \mapsto s_1, \ldots, x_n \mapsto s_n\}$ denotes ordinary capture avoiding substitution that replaces the variables with terms $s_1, \ldots, s_n$.

Definition 2.1. (Substitution of terms for metavariables) Let $\Gamma = y_1 : \overrightarrow{b_1} \cdots, y_k : \overrightarrow{b_1}$. Suppose

$$\Gamma',\, x_i^1 : a_i^1, \ldots, x_i^{n_i} : a_i^{n_i} \vdash s_i \ : \overrightarrow{b_i} \qquad (1 \le i \le k)$$

$$\textsc{m}_1 : \overrightarrow{a_1} \to \overrightarrow{b_1}, \ldots, \textsc{m}_k : \overrightarrow{a_k} \to \overrightarrow{b_k} \triangleright \Gamma \vdash e \ : \overrightarrow{c}$$

where $n_i = |\overrightarrow{a_i}|$ and $\overrightarrow{a_i} = a_i^1, \ldots, a_i^{n_i}$. Then a substitution $\ \Gamma, \Gamma' \vdash e\,[\,\overrightarrow{\textsc{m} := s}\,] \ : \overrightarrow{c}\ $ is inductively defined as follows.

$$\begin{aligned}
x\,[\,\overrightarrow{\textsc{m} := s}\,] &\triangleq x \\
\textsc{m}_i[t_1, \ldots, t_{n_i}]\,[\,\overrightarrow{\textsc{m} := s}\,] &\triangleq s_i\ \{x_i^1 \mapsto t_1\,[\,\overrightarrow{\textsc{m} := s}\,], \ldots, x_i^{n_i} \mapsto t_{n_i}\,[\,\overrightarrow{\textsc{m} := s}\,]\} \\
f(\overrightarrow{y_1}.t_1, \ldots, \overrightarrow{y_m}.t_m)\,[\,\overrightarrow{\textsc{m} := s}\,] &\triangleq f(\overrightarrow{y_1}.t_1\,[\,\overrightarrow{\textsc{m} := s}\,], \ldots, \overrightarrow{y_m}.t_m\,[\,\overrightarrow{\textsc{m} := s}\,])
\end{aligned}$$

$$\frac{y : b \in \Gamma}{\Theta \triangleright \Gamma \vdash\ y\ : b}$$

$$\frac{(\mathrm{M} : a_1, \ldots, a_m \to \overrightarrow{b}) \in \Theta \qquad \Theta \triangleright \Gamma \vdash\ t_i\ : a_i \quad (1 \le i \le m)}{\Theta \triangleright \Gamma \vdash\ \mathrm{M}[t_1, \ldots, t_m]\ :\ \overrightarrow{b}}$$

$$\frac{f : (\overrightarrow{a_1} \to \overrightarrow{b_1}), \cdots, (\overrightarrow{a_m} \to \overrightarrow{b_m}) \to \overrightarrow{c} \in \Sigma \qquad \Gamma, \overrightarrow{x_i : a_i} \vdash t_i\ :\ \overrightarrow{b_i} \quad (1 \le i \le m)}{\Theta \triangleright \Gamma \vdash\ f(\ \overrightarrow{x_1^{a_1}}.t_1, \ldots \overrightarrow{x_m^{a_m}}.t_m\ )\ :\ \overrightarrow{c}}$$

Figure 2: Typing rules of meta-terms

$$(\mathrm{Ax1})\ \frac{\Gamma', \overrightarrow{x_i : a_i} \vdash s_i\ :\ \overrightarrow{b_i} \quad (1 \le i \le k) \qquad (\mathrm{M}_1 : \overrightarrow{a_1} \to \overrightarrow{b_1}, \ldots, \mathrm{M}_k : \overrightarrow{a_k} \to \overrightarrow{b_k} \triangleright \Gamma \vdash\ t_1 = t_2\ :\ \overrightarrow{c}) \in \mathcal{E}}{\Gamma, \Gamma' \vdash t_1\,[\,\overrightarrow{\mathrm{M} := s}\,] = t_2\,[\,\overrightarrow{\mathrm{M} := s}\,]\ :\ \overrightarrow{c}}$$

$$(\mathrm{Ax2})\ \frac{\Gamma', \overrightarrow{x_i : a_i} \vdash s_i\ :\ \overrightarrow{b_i} \quad (1 \le i \le k) \qquad (\mathrm{M}_1 : \overrightarrow{a_1} \to \overrightarrow{b_1}, \ldots, \mathrm{M}_k : \overrightarrow{a_k} \to \overrightarrow{b_k} \triangleright \Gamma \vdash\ t_1 = t_2\ :\ \overrightarrow{c}) \in \mathcal{E}}{\Gamma, \Gamma' \vdash t_2\,[\,\overrightarrow{\mathrm{M} := s}\,] = t_1\,[\,\overrightarrow{\mathrm{M} := s}\,]\ :\ \overrightarrow{c}}$$

$$(\mathrm{Fun})\ \frac{f : (\overrightarrow{a_1} \to \overrightarrow{b_1}), \cdots, (\overrightarrow{a_k} \to \overrightarrow{b_k}) \to \overrightarrow{c} \in \Sigma \qquad \Gamma, \overrightarrow{x_i : a_i} \vdash t_i = t_i'\ :\ \overrightarrow{b_i} \quad (\text{some } i \text{ s.t. } 1 \le i \le k)}{\Gamma \vdash f(\overrightarrow{x_1^{a_1}}.t_1, \ldots, \overrightarrow{x_i^{a_i}}.t_i, \ldots \overrightarrow{x_1^{a_1}}.t_k) = f(\overrightarrow{x_1^{a_1}}.t_1, \ldots, \overrightarrow{x_i^{a_i}}.t_i', \ldots, \overrightarrow{x_1^{a_1}}.t_k)\ :\ \overrightarrow{c}}$$

$$(\mathrm{Ref})\ \frac{}{\Gamma \vdash t = t\ :\ \overrightarrow{c}} \qquad (\mathrm{Tra})\ \frac{\Gamma \vdash s = t\ :\ \overrightarrow{c} \qquad \Gamma \vdash t = u\ :\ \overrightarrow{c}}{\Gamma \vdash s = u\ :\ \overrightarrow{c}} \qquad (\mathrm{OSub})\ \frac{\Gamma \vdash s_i\ :\ \overrightarrow{b_i} \quad (1 \le i \le k) \qquad x_1 : b_1, \ldots, x_k : b_k \vdash t = t'\ :\ \overrightarrow{c}}{\Gamma \vdash t\,[\,\overrightarrow{x \mapsto s}\,] = t'\,[\,\overrightarrow{x \mapsto s}\,]\ :\ \overrightarrow{c}}$$

In (Ax1)(Ax2), $[\,\overrightarrow{\mathrm{M} := s}\,]$ denotes $[\,\mathrm{M}_1 := s_1, \ldots, \mathrm{M}_k := s_k\,]$.
In (OSub), $[\,\overrightarrow{x \mapsto s}\,]$ denotes $[\,x_1 \mapsto s_1, \ldots, x_k \mapsto s_k\,]$.

Figure 3: Cartesian second-order equational logic

---

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

Remark 2.2. The syntactic structure of meta-terms and substitution for abstract syntax with variable binding was introduced by Aczel [Acz78]. This formal language allowed him to consider a general framework of rewrite rules for calculi with variable binding. This influenced Klop's rewrite system of Combinatory Reduction System (CRS) [Klo80]. A second-order substitution in the sense of Courcelle [Cou83] is very similar to a substitution for metavariables but there are no variable binders in the language used in [Cou83] where function symbols are the targets of replacements (instead of metavariables in our framework).

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 from a given set $\mathcal{E}$ of equations, regarded as *axioms*. The inference system of equational logic is given in Fig. 3. Note that the symmetry rule

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

is derivable because of symmetry of (Ax1) and (Ax2).

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

Empty sequence $\langle\rangle \quad : ()$	Tuple $\langle -, \cdots, - \rangle : (\overrightarrow{c_1}), \ldots, (\overrightarrow{c_n}) \to \overrightarrow{c_1}, \ldots, \overrightarrow{c_n}$
Cycle constructor $\mathsf{cy}^{\lvert \overrightarrow{c} \rvert} : (\overrightarrow{c} \to \overrightarrow{c}) \to \overrightarrow{c}$	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 Σ includes Σ_{def} in this paper. 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_c and axioms $\mathcal{E}$, where every $f \in \Sigma_c$ is of the form

$$f : \overrightarrow{a}, c, \ldots, c \to c,$$

where $\overrightarrow{a}$ are types other than c (which may be empty), and any equation in $\mathcal{E}$ is built from Σ_c-terms.

2.3. **Instance (1): Cyclic Lists modulo Bisimulation.** We will present an algebraic formulation of cyclic datatypes. By a cyclic datatype, we mean an algebraic datatype having the cycle construct cy satisfying the axioms that characterise cyclicity. The first example is the datatype of natural numbers. It has already been defined as `CNat` in Introduction as a pseudo code. We now give a formal definition using a datatype declaration. The datatype declaration for `CNat` is given by $(\mathtt{CNat}, \Sigma_{\mathtt{CNat}}, \mathsf{AxCy})$ where $\Sigma_{\mathtt{CNat}}$ is

$$\mathtt{0} : \mathtt{CNat}, \qquad \mathtt{S} : \mathtt{CNat} \to \mathtt{CNat}$$

and the axioms AxCy are given in Fig. 4.

The second example is the datatype of cyclic lists. It has already been defined as `CList` in Introduction as a pseudo code. 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 given by

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

and the axioms AxCy are given in Fig. 4. Note that $\mathtt{CList}_a$ has also the default constructors, thus one can form a 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 often 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

Axioms AxCy for cycles

$$
\begin{array}{llll}
\text{(sub)} & \begin{array}{l}\mathrm{T}: \vec{a} \to \vec{c},\\ \mathrm{S}_1,\dots,\mathrm{S}_n : \vec{a}\end{array} & \rhd \vdash \quad (\vec{y}.\mathrm{T}[\vec{y}]) \diamond \langle \mathrm{S}_1,\dots,\mathrm{S}_n\rangle = \mathrm{T}[\mathrm{S}_1,\dots,\mathrm{S}_n] & : \vec{c} \\
\text{(SP)} & \mathrm{T} : \vec{c} & \rhd \vdash \langle (\vec{y}.y_1) \diamond \mathrm{T}, \dots, (\vec{y}.y_n) \diamond \mathrm{T}\rangle = \mathrm{T} & : \vec{c} \\
(\text{dinat}_1) & \begin{array}{l}\mathrm{S}: a \to c,\\ \mathrm{T}: c \to a\end{array} & \rhd \vdash \quad \mathsf{cy}(x.\mathrm{S}[\mathrm{T}[x]]) = \mathrm{S}[\mathsf{cy}(z.\mathrm{T}[\mathrm{S}[z]])] & : c \\
(\text{dinat}_n) & \begin{array}{l}\mathrm{S}: \vec{a} \to \vec{c},\\ \mathrm{T}: \vec{c} \to \vec{a}\end{array} & \rhd \vdash \quad \mathsf{cy}(\vec{x}.\mathrm{S}[\mathrm{T}[\vec{x}]]) = (\vec{z}.\mathrm{S}[\vec{z}]) \diamond \mathsf{cy}(\vec{z}.\mathrm{T}[\mathrm{S}[\vec{z}]]) & : \vec{c} \\
\text{(Bekič)} & \begin{array}{l}\mathrm{T}: \vec{c},\vec{a} \to \vec{c},\\ \mathrm{S}: \vec{c},\vec{a} \to \vec{a}\end{array} & \rhd \vdash \mathsf{cy}(\vec{x},\vec{y}.\langle \widehat{\mathrm{T}}, \widehat{\mathrm{S}}\rangle) = \left\langle \begin{array}{l} \mathsf{cy}(\vec{x}.(\vec{y}.\widehat{\mathrm{T}}) \diamond \mathsf{cy}(\vec{y}.\widehat{\mathrm{S}})),\\ \mathsf{cy}(\vec{y}.(\vec{x}.\widehat{\mathrm{S}}) \diamond \mathsf{cy}(\vec{x}.(\vec{y}.\widehat{\mathrm{T}}) \diamond \mathsf{cy}(\vec{y}.\widehat{\mathrm{S}}))) \end{array}\right\rangle & : \vec{c}, \vec{a} \\
\text{(CI)} & \mathrm{T} : a^m \to a & \rhd \vdash \quad \mathsf{cy}(\vec{y}.\langle \mathrm{T}[\rho_1],\dots,\mathrm{T}[\rho_m]\rangle) = \langle \mathsf{cy}(y.\tilde{\mathrm{T}}),\dots,\mathsf{cy}(y.\tilde{\mathrm{T}})\rangle & : a^m
\end{array}
$$

In (dinat_n), $|\vec{c}| = m > 1$ and $\mathrm{S}[\mathrm{T}[\vec{x}]]$ is short for $\mathrm{S}[(\vec{y}.y_1) \diamond \mathrm{T}[\vec{x}],\dots,(\vec{y}.y_m) \diamond \mathrm{T}[\vec{x}]]$. Similarly for $\mathrm{T}[\mathrm{S}[\vec{z}]]$. In (Bekič), $\widehat{\mathrm{T}}$ and $\widehat{\mathrm{S}}$ are short for $\mathrm{T}[\vec{x},\vec{y}]$ and $\mathrm{S}[\vec{x},\vec{y}]$, respectively. In (CI), $\rho_i = y_{i_1},\dots,y_{i_m}$, where $i_1,\dots,i_m \in \{1,\dots,m\}$, $\tilde{\mathrm{T}}$ is short for $\mathrm{T}[y,\dots,y]$.

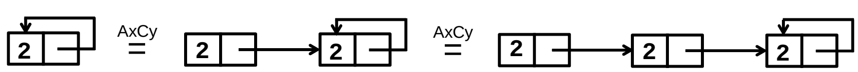
Axioms $\mathsf{AxBr}([\,],+)$ for branching

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

Note that the axiom $\mathsf{AxBr}([\,],+)$ is parameterised by the function symbols $[\,],+$. In general, writing $\mathsf{AxBr}(\nu,\mu)$ where $\nu : c,\ \mu : c,c \to c \in \Sigma_{\mathsf{con}}$, we mean the set of axioms obtained from the above axioms by replacing $[\,]$ with ν, and $+$ with μ.

Figure 4: Axioms

[BÉ93]. The equational theory generated by AxCy captures the intended meaning of cyclic lists. For example, the following are identified as the same cyclic list:



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

These equalities come from the *fixed point law* of cy:

$$\text{(fix)} \quad \mathsf{cy}(x.\mathrm{S}[x]) \quad = \quad \mathrm{S}[\,\mathsf{cy}(x.\mathrm{S}[x])\,]$$

which is an instance of the axiom (dinat) when T is identity (i.e., $\mathrm{T} \mapsto x$ for $x : c \vdash x \ : c$).

2.4. **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 [BÉ93, Has97, SP00]. Bekič law is well-known in denotational semantics (cf. [Win93, §10.1]) to calculate the fixed point of a pair of continuous functionns. Conway operators are also arisen in work independently of Hyland and Hasegawa [Has97], who established a connection with the notion of traced cartesian categories [JSV96]. 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 [BÉ93, SP00], which ensures that all equalities that hold in the cpo semantics do hold. See also [SP00, Section 2] for a useful overview about this. The equality generated by AxCy is bisimulation on cyclic lists. This is included in the equality of cyclic sharing trees given in the next subsection.

2.5. **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 $f_1, \ldots, f_n$'s part denotes various unary function symbols such as `a,b,c,p,q,`....

$$
\begin{array}{l}
\texttt{ctype CTree where} \\
\quad f_1 \; \texttt{: CTree} \to \texttt{CTree} \\
\quad \;\vdots \\
\quad f_n \; \texttt{: CTree} \to \texttt{CTree} \\
\quad [\,] \; \texttt{: CTree} \\
\quad \texttt{+} \; \texttt{: CTree,CTree} \to \texttt{CTree} \\
\texttt{with axioms } \mathsf{AxCy}\texttt{,}\mathsf{AxBr}\texttt{(}[\,]\texttt{,+)}
\end{array}
$$

Formally, it is expressed as the datatype declaration

$$(\texttt{CTree}, \{f_1, \ldots, f_n, [\,], +\}, \; \mathsf{AxCy} \cup \mathsf{AxBr}([\,], +)).$$

The binary operator `+` denotes a branch. For example, one can write `b([])+c([])` (cf. Fig. 5 (A)). The datatype can express sharing by the constructor $\diamond$ of composition:

$$\texttt{(x.a(b(x) + c(x)))} \diamond \texttt{p([])}$$

(cf. Fig. 5 (F)). Note that the first argument of composition $\diamond$ has a binder (e.g. `x.`), which indicates a 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* [CGZ05], and arbitrary rooted directed graphs (cf. Fig. 5 (B)(E)).

We denote by $\sim$ the equivalence relation generated by the axioms $\mathsf{AxCy}, \mathsf{AxBr}([\,], +)$ in Fig. 4. 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. The branch $+$ is associative, commutative and idempotent (cf. Fig. 5 (C)), thus nested $+$ can be seen as an n-ary branch (cf. Fig. 5 (D)). Moreover, a shared term and its unfolding are also identified by $\sim$ because of the axiom (sub) (cf. Fig. 5 (F)). The axiom (sub) is similar to the β-reduction in the λ-calculus.

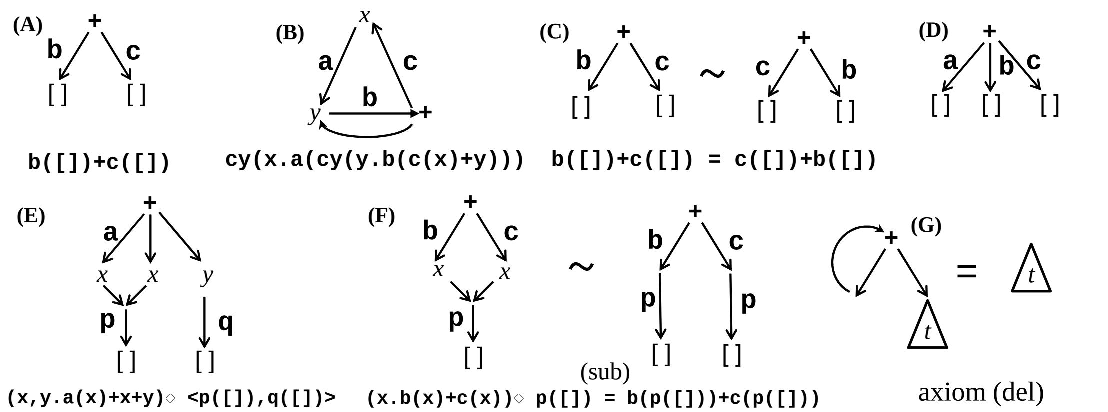


Figure 5: Examples of cyclic sharing trees

2.6. **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 [Mil84]. Infinite unfolding of them are synchronization trees [BÉ93]. 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 [BÉ93, Mil84] or ([HMA] Appendix)). Now we see that (C),(F) and (G) are examples of bisimulation. Actually, the axioms in Fig. 4 are sound for bisimulation, i.e., for each axiom, the left and the right-hand sides are bisimilar. Moreover, it is complete.

Proposition 2.3. **(**[Ham15]**,(**[HMA]**§5.3))** $\Gamma \vdash s = t$: `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 a second-order representation of Bloom and Ésik's complete equational axioms of bisimulation [BÉ93]. A crucial fact is that bisimulation $s \sim t$ is decidable [BÉ93, BFS00]. There is also an efficient algorithm for checking bisimulation, e.g. [DPP04]. Hence, cyclic datatypes with the axioms AxCy, or the axioms $\mathsf{AxCy} \cup \mathsf{AxBr}$ are computationally feasible. For example, equality on cyclic structures such as the one we have seen in Fig. 1 can be checked efficiently.

There are many other instances of cyclic datatypes, some of which will be given in §7.

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. 3.9 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 [BÉ93]. In particular iteration categories [Ési99] are suitable for our purpose, which are *traced cartesian categories* [JSV96, Has97] satisfying the *commutative identities* axiom [BÉ93]. We write **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.

Definition 3.1. [Ési99, BÉ93] 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 [BÉ93]

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

where
- $f : A \times X^m \to X$
- $\Delta_m \triangleq \langle \mathrm{id}_X, \cdots, \mathrm{id}_X \rangle : X \to X^m$ is the diagonal
- $\rho_i : X^m \to X^m$ such that $\rho_i = \langle q_{i1}, \ldots, q_{im} \rangle$ where each q_{ij} is one of projections $\pi_1, \ldots, \pi_m : X^m \to X$ (see also [SP00]).

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

Remark 3.2. It is well-known that there are several equivalent axiomatisations of the Coway operator, cf. [BÉ93, §6.8][Has97, §7.1] and [SP00]. A frequently used axiomatisation is a natural and dinatural operator satisfying the Bekič law, which is what the axioms AxCy in Fig. 4 state.

A typical example of iteration category is the category of **CPO** complete partial orders (cpos) with bottom and continuous functions [BÉ93, Has97], where the least fixed point operator is a Conway operator.

Definition 3.3. 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 $f : (\overrightarrow{a_1} \to \overrightarrow{b_1}), \ldots, (\overrightarrow{a_m} \to \overrightarrow{b_m}) \to \overrightarrow{c}$, a function

$$[\![f]\!]^M_A : \mathcal{C}(A \times [\![\overrightarrow{a_1}]\!], [\![\overrightarrow{b_1}]\!]) \times \cdots \times \mathcal{C}(A \times [\![\overrightarrow{a_n}]\!], [\![\overrightarrow{b_n}]\!]) \longrightarrow \mathcal{C}(A, [\![\overrightarrow{c}]\!]) \tag{3.1}$$

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.

3.1. 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

$$[\![\Gamma \vdash y_i \;\; : c]\!]^M = \pi_i : [\![\Gamma]\!] \to [\![c]\!]$$

$$[\![\Gamma \vdash f(\, \overrightarrow{x_1^{a_1}}.t_1, \ldots \overrightarrow{x_n^{a_n}}.t_n \,) \;\; : \overrightarrow{c}]\!]^M = [\![f]\!]^M_{[\![\Gamma]\!]}(\, [\![\Gamma, \overrightarrow{x_1 : a_1} \vdash t_1 \;\; : \overrightarrow{b_1}]\!]^M, \ldots, [\![\Gamma, \overrightarrow{x_n : a_n} \vdash t_n \;\; : \overrightarrow{b_n}]\!]^M).$$

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

$$
\begin{aligned}
&[\![\langle\,\rangle]\!]_A^M : \mathcal{C}(A,\mathbf{1}) \to \mathcal{C}(A,\mathbf{1})\\
&[\![\langle\,\rangle]\!]_A^M(t) = t\\
&[\![\langle -,\dots,-\rangle]\!]_A^M : \mathcal{C}(A,[\![\overrightarrow{c_1}]\!]) \times \cdots \times \mathcal{C}(A,[\![\overrightarrow{c_n}]\!]) \to \mathcal{C}(A,[\![\overrightarrow{c_1}]\!] \times \cdots \times [\![\overrightarrow{c_n}]\!])\\
&[\![\langle -,\dots,-\rangle]\!]_A^M(t_1,\dots,t_n) = \langle t_1,\dots,t_n\rangle\\
&[\![\diamond]\!]_A^M : \mathcal{C}(A\times[\![\overrightarrow{a}]\!],[\![\overrightarrow{c}]\!]) \times \mathcal{C}(A,[\![\overrightarrow{a}]\!]) \to \mathcal{C}(A,[\![\overrightarrow{c}]\!])\\
&[\![\diamond]\!]_A^M(t,s) = t\circ\langle \mathrm{id}_A, s\rangle\\
&[\![\mathsf{cy}]\!]_A^M : \mathcal{C}(A\times[\![\overrightarrow{c}]\!],[\![\overrightarrow{c}]\!]) \to \mathcal{C}(A,[\![\overrightarrow{c}]\!])\\
&[\![\mathsf{cy}]\!]_A^M(t) = t^\dagger
\end{aligned}
$$

We say a Σ-structure M *satisfies* an equation $\Gamma \vdash s = t \quad : c$, if $[\![s]\!]^M = [\![t]\!]^M$ holds. Importantly, every Σ_{def}-structure satisfies the axioms AxCy because $\mathcal{C}$ is an iteration category.

Definition 3.4. A $(\Sigma,\mathcal{E})$*-structure* is a Σ-structure M in $\mathcal{C}$ satisfying all equations in a set $\mathcal{E}$ of axioms. 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 $(c,\Sigma_c,\mathcal{E})$ consists of a $(\Sigma_c,\mathcal{E})$-structure M with a family of morphisms of $\mathcal{C}$:

$$\alpha \;\triangleq\; (\; (\![f]\!)^M : [\![b_1]\!] \times \ldots \times [\![b_n]\!] \to [\![c]\!] \;)_{f:b_1,\ldots,b_n\to c\in\Sigma_c}.$$

Note that it induces the interpretation of every function symbol $f\in\Sigma_c$ given by

$$[\![f]\!]_A^M(t_1,\dots,t_n) = (\![f]\!)^M \circ \langle t_1,\dots,t_n\rangle$$

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 3.5. **(The datatype $\texttt{CList}_a$ of cyclic lists of type a)** We give a $\texttt{CList}_a$-structure in the iteration category **CPO** via the standard initial algebra characterisation of datatypes in the category $\mathbf{CPO}_\perp$ of cpos and strict continuous functions [AJ94]. We write that 1 is the cpo $\{\perp\}$ (which is the initial object of $\mathbf{CPO}_\perp$), where $\perp$ is the least element, $\oplus$ is the coalesced sum, and $(-)_\perp$ is lifting.

Let A be the interpretation $[\![a]\!]^L$ of a base type a in **CPO**. We construct an initial algebra L of the functor F on $\mathbf{CPO}_\perp$ defined by $F(X) = 1_\perp \oplus (A\times X)_\perp$ by using the initial algebra construction [SP82], i.e., taking the colimit of ω-chain

$$1 \xrightarrow{\;!\;} F(1) \xrightarrow{F(!)} F^2(1) \xrightarrow{F^2(!)} \cdots$$

we have a cpo $L\cong F(L)$, consisting of finite and infinite possibly partial lists with continuous functions $\mathsf{nil}:1\to L$ and $\mathsf{cons}:A\times L\to L$. The order of the cpo is the usual pointwise ordering $\perp\sqsubseteq t$, and $(t_1,t_2)\sqsubseteq(t_1',t_2')$ if $t_1\sqsubseteq t_1'$ and $t_2\sqsubseteq t_2'$, etc. Then we have a $\texttt{CList}_a$-structure $(L,((\![[\,]]\!)^L,(\![::]\!)^L))$ defined by

$$(\![[\,]]\!)^L \;=\; \mathsf{nil}, \qquad (\![::]\!)^L \;=\; \mathsf{cons}$$

and all the axioms in AxCy hold in **CPO**, since these are now well-known propeties of cpos [Win93, BÉ93, SP00]. Note that this construction is the same as modelling the lazy list datatype, hence this also shows that our framework covers modelling lazy datatypes as in Haskell.

Example 3.6. (The cyclic tree type `CTree`) A `CTree`-structure in an iteration category $\mathcal{C}$ is given by a $(\Sigma_{\mathsf{def}}, \mathsf{AxCy} \cup \mathsf{AxBr}([\,], +))$-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

$$(\![[\,]]\!)^M \ = \ \eta, \qquad (\![+]\!)^M \ = \ \mu.$$

It satisfies all axioms of $\mathsf{AxBr}([\,], +)$. Note that every `CTree`-structure is a *degenerated commutative bialgebra* (cf. [FC13]) in a cartesian category $\mathcal{C}$, i.e., N forms also a comonoid $(N, !, \Delta)$ that satisfies the compatibility

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

The last equation is by $\mu \circ \Delta \ = \ \mu \circ \langle \mathrm{id}, \mathrm{id} \rangle \ = \ \mu \circ \langle \mathrm{id}, (\mu)^{\dagger} \rangle \overset{\text{(dinat)}}{=} (\mu)^{\dagger} \ = \ \mathrm{id}$. Thus, a `CTree`-structure models branch (by μ) and sharing by (Δ) 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. 3), 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 term judgments modulo $=_{\mathcal{E}}$ 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 3.7. $\mathbf{Tm}(\mathcal{E})$ 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

$$[\![f]\!]^{\mathrm{U}}_{\overrightarrow{a}} : \mathbf{Tm}(\mathcal{E})((\overrightarrow{a}, \overrightarrow{a_1}), \overrightarrow{b_1}) \times \cdots \times \mathbf{Tm}(\mathcal{E})((\overrightarrow{a}, \overrightarrow{a_n}), \overrightarrow{b_n}) \longrightarrow \mathbf{Tm}(\mathcal{E})(\overrightarrow{a}, \overrightarrow{c})$$
$$[\![f]\!]^{\mathrm{U}}_{\overrightarrow{a}}([t_1]_{\mathcal{E}}, \ldots, [t_n]_{\mathcal{E}}) \triangleq [f(\overrightarrow{x_1}.t_1, \ldots, \overrightarrow{x_n}.t_n)]_{\mathcal{E}}$$

for $f : (\overrightarrow{a_1} \to \overrightarrow{b_1}), \ldots, (\overrightarrow{a_m} \to \overrightarrow{b_m}) \to \overrightarrow{c} \in \Sigma$ and base types $\overrightarrow{a}$. This is well-defined because $=_{\mathcal{E}}$ is a congruence. 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 op.: $([\Gamma, \overrightarrow{x : c} \vdash t \ : \overrightarrow{c}]_{\mathcal{E}})^{\dagger} = [\Gamma \vdash \mathsf{cy}(\overrightarrow{x^c}.t) \ : \overrightarrow{c}]_{\mathcal{E}}$
- projections: $[x_1 : c_1, x_2 : c_2 \vdash x_i \ : c_i]_{\mathcal{E}}$

Then these data turn $\mathbf{Tm}(\mathcal{E})$ into an iteration category, and moreover, U forms a $(\Sigma, \mathcal{E})$-structure because of the axioms $\mathcal{E}$ for each $c \in \mathcal{B}$. Moreover,

$$(\ [\![c]\!]^{\mathrm{U}},\ (f : [\![\overrightarrow{b_1}]\!] \times \cdots \times [\![\overrightarrow{b_m}]\!] \to [\![c]\!])_{f : \overrightarrow{b_1}, \ldots, \overrightarrow{b_m} \to c \in \Sigma_c}\ )$$

is a c-structure. Note that $[\![\overrightarrow{b_i}]\!] = \overrightarrow{b_i}$ in $\mathbf{Tm}(\mathcal{E})$. ∎

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

Theorem 3.8. **(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 3.9. For a $(\Sigma, \mathcal{E})$-structure M in an iteration category $\mathcal{C}$, there exists a unique iteration functor $\Psi^M : \mathbf{Tm}(\mathcal{E}) \longrightarrow \mathcal{C}$ that preserves $(\Sigma, \mathcal{E})$-structures. Pictorially, it is expressed as the following diagram, where Tm denotes the set of all terms (without quotient).

$$\begin{tikzcd} \mathrm{Tm} \arrow[r, "{[\![-]\!]^{\mathrm{U}}}"] \arrow[d, "{[\![-]\!]^M}"'] & \mathbf{Tm}(\mathcal{E}) \arrow[dl, "\Psi^M"] \\ \mathcal{C} \end{tikzcd}$$

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

$$\begin{array}{ll}
\Psi(\ [\Gamma \vdash y_i \ : c]_{\mathcal{E}} &) = \ \pi_i \\
\Psi(\ [\Gamma \vdash \langle\rangle \ : ()]_{\mathcal{E}} &) = \ ! \\
\Psi(\ [\Gamma \vdash \langle s\, , \, t\rangle \ : \overrightarrow{c_1}, \overrightarrow{c_2}]_{\mathcal{E}} &) = \langle \Psi[\Gamma \vdash s \ : \overrightarrow{c_1}]_{\mathcal{E}}, \Psi[\Gamma \vdash t \ : \overrightarrow{c_2}]_{\mathcal{E}} \rangle \\
\Psi(\ [\Gamma \vdash \mathsf{cy}(\overrightarrow{x^c}.t) \ : \overrightarrow{c}]_{\mathcal{E}} &) = (\Psi[\Gamma, \overrightarrow{x : c} \vdash t \ : \overrightarrow{c}]_{\mathcal{E}})^{\dagger} \\
\Psi(\ [\Gamma \vdash f(\ \overrightarrow{x_1^{a_1}}.t_1, \ldots, \overrightarrow{x_m^{a_m}}.t_m) \ : c]_{\mathcal{E}}) & \\
\quad = [\![f]\!]^M_{[\![\Gamma]\!]}(\Psi[\Gamma, \overrightarrow{x_1 : a_1} \vdash t_1 \ : b_1]_{\mathcal{E}}, \ldots, \Psi[\Gamma, \overrightarrow{x_m : a_m} \vdash t_m \ : b_m]_{\mathcal{E}}) & \\
\Psi(\ [\Gamma \vdash (\overrightarrow{x^b}.t) \diamond s \ : c]_{\mathcal{E}} &) = \Psi[\Gamma, \overrightarrow{x : b}, \vdash t \ : c]_{\mathcal{E}} \circ \langle \mathrm{id}_{[\![\Gamma]\!]}, \Psi[\Gamma \vdash s \ : \overrightarrow{b}]_{\mathcal{E}} \rangle
\end{array} \tag{3.2}$$

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 Datatypes

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 another cyclic datatype b by the functor Ψ^M. Let $(c, \Sigma_c, \mathcal{E}_c)$ and $(b, \Sigma_b, \mathcal{E}_b)$ be datatype declarations. Let $\mathcal{E}$ be the set of axioms collecting the axioms of all datatype declarations for types in $\mathcal{B}$, which includes AxCy (and AxBr if the datatype suppose it) for every datatype. Hence $\mathcal{E} \supseteq \mathcal{E}_b \cup \mathcal{E}_c$. We define a $(\Sigma_c, \mathcal{E})$-structure (M, α) in $\mathbf{Tm}(\mathcal{E})$ by

$$[\![c]\!]^M = b$$

and $[\![a]\!]^M = a$ for $a \neq c$. We write the arrow part function Ψ^M on hom-sets as the fold, i.e.,

$$\mathbf{fold}^c_b(\alpha) : \mathbf{Tm}(\mathcal{E})(\overrightarrow{c},\ c) \longrightarrow \mathbf{Tm}(\mathcal{E})([\![\overrightarrow{c}]\!],\ b). \tag{4.1}$$

where $\overrightarrow{c} = c_1, \ldots, c_n$ (N.B. some of c_i may be c).

4.1. **Axiomatising fold as a second-order algebraic theory.** The **fold** is a function on equivalence classes of term judgments modulo $\mathcal{E}$ characterised by (3.2). Equivalently, we regard it as a function on terms (or term judgments) that preserves $=_{\mathcal{E}}$, i.e.,

$$s =_{\mathcal{E}} t \quad \Rightarrow \quad \mathbf{fold}^c_b(\alpha)(s) \ =_{\mathcal{E}} \ \mathbf{fold}^c_b(\alpha)(t).$$

In this subsection, we axiomatise the function $\mathbf{fold}^c_b$ as the laws of fold within second-order equational logic using (3.2).

Axiomatising 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 $(\!|f|\!)^M = [e_f]_{\mathcal{E}}$. Note that $[\![c]\!] = b$. 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.

Axiomatising fold. We next axiomatise the fold operation as a second-order algebraic theory. The type of fold may be chosen as

$$\mathsf{fold}^c_b : ([\![\overrightarrow{a_1}]\!] \to b), \ldots, ([\![\overrightarrow{a_m}]\!] \to b), (\overrightarrow{c} \to c) \to ([\![\overrightarrow{c}]\!] \to b), \tag{4.2}$$

where the first m-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 codomain $([\![\overrightarrow{c}]\!] \to b)$ is inappropriate. We resolve this by simply uncurrying. We axiomatise the fold as the function symbol of the type

$$\mathsf{fold}^c_b : ([\![\overrightarrow{a_1}]\!] \to b), \ \ldots, ([\![\overrightarrow{a_m}]\!] \to b), (\overrightarrow{c} \to c), [\![\overrightarrow{c}]\!] \longrightarrow b.$$

and we will write a term of it using the notation

$$\mathsf{fold}^c_b(\,\overrightarrow{x_1}.e_1, \ldots, \overrightarrow{x_m}.e_m,\ \overrightarrow{y}.\,t\,;\ \overrightarrow{y}\,)$$

where each e_i corresponds to $(\!|f_i|\!)^M$ for $f_i \in \Sigma_c$ in α. The last two arguments $\overrightarrow{y}.\,t$ and $\overrightarrow{y}$ may need explanation. The first "$\overrightarrow{y}.\,t$" describes a term with variable binding "$\overrightarrow{y}.$" of types $\overrightarrow{c}$, but the second "$\overrightarrow{y}$" are free variables of types $[\![\overrightarrow{c}]\!]$, which are different from the first bound variables, but have the same length. We may write simply

$$\mathsf{fold}^c_b(\,\overrightarrow{x_1}.e_1, \ldots, \overrightarrow{x_m}.e_m,\ t\,)$$

by omitting the parameters after ";", when $|\overrightarrow{y}| = 0$. This is a formalisation of the mathematical expression $\mathbf{fold}^c_b(\alpha)([\overline{y : c} \vdash t \ \ : c]_{\mathcal{E}})$ at the level of semantics as a syntactic term.

In general, we can consider $\mathsf{fold}^{\overrightarrow{c}}_{\overrightarrow{b}}$ as the fold from $\overrightarrow{c}$ to $\overrightarrow{b}$. Hereafter, we assume that any signature Σ is divided into the default signature Σ_{def}, a *signature for constructors* Σ_{con}, and fold's:

$$\Sigma = \Sigma_{\mathsf{def}} \cup \Sigma_{\mathsf{con}} \cup \{\mathsf{fold}^{\overrightarrow{c}}_{\overrightarrow{b}} \mid \overrightarrow{c}, \overrightarrow{b} \in \mathcal{B}\}.$$

In other words, we assume that any function symbol other than a default constructor or fold is an element of Σ_{con}.

In Fig. 6, we give the axioms FOLD, which axiomatise fold by using the characterisation (3.2) in case of a particular category $\mathcal{C} = \mathbf{Tm}(\mathcal{E})$ and a c-structure as a second-order algebraic theory. To ease understanding, we explicitly describe an instance of (5) as (5') for the case $c = \texttt{CList}$, $d = (::)$. Note that there is no case of $\mathsf{fold}^c_b(E,\ \overrightarrow{y}.\,z;\ \overrightarrow{y})$ for $z \notin \{\,\overrightarrow{y}\,\}$.

$$(1)\ \mathsf{fold}^c_b(E,\ \overrightarrow{y^c}.\, y^c_i \quad ;\ \overrightarrow{y^b}\,) = {y_i}^b \qquad (\text{for } y^c_i \in \{y^{c_1}_1, \dots, y^{c_n}_n\})$$

$$(2)\ \mathsf{fold}^c_{()}(E,\ \overrightarrow{y}.\, \langle\,\rangle \quad ;\ \overrightarrow{y}\,) = \langle\,\rangle$$

$$(3)\ \mathsf{fold}^{\overrightarrow{c_1},\overrightarrow{c_2}}_{\overrightarrow{b_1},\overrightarrow{b_1}}(E,\ \overrightarrow{y}.\, \langle \mathrm{S}[\overrightarrow{y}], \mathrm{T}[\overrightarrow{y}]\rangle \ ;\ \overrightarrow{y}\,) = \langle \mathsf{fold}^{\overrightarrow{c_1}}_{\overrightarrow{b_1}}(E,\ \overrightarrow{y}.\, \mathrm{S}[\overrightarrow{y}];\ \overrightarrow{y}\,),\ \mathsf{fold}^{\overrightarrow{c_1}}_{\overrightarrow{b_2}}(E,\ \overrightarrow{y}.\, \mathrm{T}[\overrightarrow{y}];\ \overrightarrow{y}\,)\rangle$$

$$(4)\ \mathsf{fold}^{\overrightarrow{c}}_{\overrightarrow{b}}(E,\ \overrightarrow{y}.\, \mathsf{cy}(\overrightarrow{x}.\mathrm{T}[\overrightarrow{y}, \overrightarrow{x}])\ ;\ \overrightarrow{y}\,) = \mathsf{cy}(\overrightarrow{x}.\mathsf{fold}^{\overrightarrow{c}}_{\overrightarrow{b}}(E,\ \overrightarrow{y}, \overrightarrow{x}.\, \mathrm{T}[\overrightarrow{y}, \overrightarrow{x}];\ \overrightarrow{y}, \overrightarrow{x}\,))$$

$$(5)\ \mathsf{fold}^c_b(E,\ \overrightarrow{y}.\, d(\overrightarrow{\mathrm{A}}, \mathrm{T}_1[\overrightarrow{y}], \dots, \mathrm{T}_n[\overrightarrow{y}]);\ \overrightarrow{y}\,) = (\overrightarrow{x}.\mathrm{E}_d[\overrightarrow{\mathrm{A}}, \overrightarrow{x}]) \diamond \langle \mathsf{fold}^{c_1}_{b_1}(E,\ \overrightarrow{y}.\, \mathrm{T}_1[\overrightarrow{y}];\ \overrightarrow{y}\,), \dots\rangle$$

$$(5')\ \mathsf{fold}^{\mathtt{CList}}_b(E,\ \overrightarrow{y}.\, \mathrm{A} :: \mathrm{T}[\overrightarrow{y}] \quad ;\ \overrightarrow{y}\,) = (x.\mathrm{E}[\mathrm{A}, x]) \diamond\ \mathsf{fold}^{\mathtt{CList}}_b(E,\ \overrightarrow{y}.\, \mathrm{T}[\overrightarrow{y}];\ \overrightarrow{y}\,))$$

$$(6)\ \mathsf{fold}^c_b(E,\ \overrightarrow{y}.\, (\overrightarrow{x}.\mathrm{T}[\overrightarrow{x}]) \diamond \mathrm{S}[\overrightarrow{y}]\ ;\ \overrightarrow{y}\,) = (\overrightarrow{x}.\mathsf{fold}^c_b(E,\ \overrightarrow{x}.\, \mathrm{T}[\overrightarrow{x}];\ \overrightarrow{x}\,))\ \diamond\ \mathsf{fold}^{\overrightarrow{d}}_{\overrightarrow{a'}}(E,\ \overrightarrow{y}.\, \mathrm{S}[\overrightarrow{y}];\ \overrightarrow{y}\,)$$

Here E is a sequence $(\overrightarrow{z}, \overrightarrow{x}.\mathrm{E}_d[\overrightarrow{z}, \overrightarrow{x}])_{d\in\Sigma_c}$ of metavariables and $d \in \Sigma_c$. (5') is an instance of (5) for explanation. 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 6: Second-order algebraic theory FOLD

$$(1\mathrm{r})\ \mathsf{fold}^c_b(E,\ \overrightarrow{y}.\, \mathsf{v}(y_i) \quad ;\ \overrightarrow{y}\,) \to \mathsf{v}(y_i)$$

$$(2\mathrm{r})\ \mathsf{fold}^c_{()}(E,\ \overrightarrow{y}.\, \langle\,\rangle \quad ;\ \overrightarrow{y}\,) \to \langle\,\rangle$$

$$(3\mathrm{r})\ \mathsf{fold}^{\overrightarrow{c_1},\overrightarrow{c_2}}_{\overrightarrow{b_1},\overrightarrow{b_1}}(E,\ \overrightarrow{y}.\, \langle \mathrm{S}[\overrightarrow{y}], \mathrm{T}[\overrightarrow{y}]\rangle \quad ;\ \overrightarrow{y}\,) \to \langle \mathsf{fold}^{\overrightarrow{c_1}}_{\overrightarrow{b_1}}(E,\ \overrightarrow{y}.\, \mathrm{S}[\overrightarrow{y}];\ \overrightarrow{y}\,),\ \mathsf{fold}^{\overrightarrow{c_1}}_{\overrightarrow{b_2}}(E,\ \overrightarrow{y}.\, \mathrm{T}[\overrightarrow{y}];\ \overrightarrow{y}\,)\rangle$$

$$(4\mathrm{r})\ \mathsf{fold}^{\overrightarrow{c}}_{\overrightarrow{b}}(E,\ \overrightarrow{y}.\, \mathsf{cy}(\overrightarrow{x}.\mathrm{T}[\overrightarrow{y}, \overrightarrow{x}]) \quad ;\ \overrightarrow{y}\,) \to \mathsf{cy}(\overrightarrow{x}.\mathsf{fold}^{\overrightarrow{c}}_{\overrightarrow{b}}(E,\ \overrightarrow{y}, \overrightarrow{x}.\, \mathrm{T}[\overrightarrow{y}, \overrightarrow{x}];\ \overrightarrow{y}, \overrightarrow{x}\,))$$

$$(5\mathrm{r})\ \mathsf{fold}^c_b(E,\ \overrightarrow{y}.\, d(\overrightarrow{\mathrm{A}}, \mathrm{T}_1[\overrightarrow{y}], \dots, \mathrm{T}_n[\overrightarrow{y}]);\ \overrightarrow{y}\,) \to \mathrm{E}_d[\overrightarrow{\mathrm{A}}, \mathsf{fold}^{c_1}_{b_1}(E,\ \overrightarrow{y}.\, \mathrm{T}_1[\overrightarrow{y}];\ \overrightarrow{y}\,), \dots \\ \mathsf{fold}^{c_n}_{b_n}(E,\ \overrightarrow{y}.\, \mathrm{T}_n[\overrightarrow{y}];\ \overrightarrow{y}\,)]$$

Composition

$$(7\mathrm{r})\ (\overrightarrow{y}.\, \mathrm{T}[\overrightarrow{y}]) \diamond \langle\, \overrightarrow{\mathrm{S}}\,\rangle \qquad \to \mathrm{T}[\overrightarrow{s}\,]$$

Figure 7: Second-order rewrite system FOLDr

Bekič

$$(10\mathrm{r})\quad \mathsf{cy}^{m+n}(\overrightarrow{x}, \overrightarrow{y}.\, \langle\, \widehat{\mathrm{T}},\ \widehat{\mathrm{S}}\,\rangle) \to \langle\, \mathsf{cy}^m(\overrightarrow{x}.\, (\overrightarrow{y}.\widehat{\mathrm{T}}) \diamond \mathsf{cy}^n(\overrightarrow{y}.\widehat{\mathrm{S}})), \\ \mathsf{cy}^n(\overrightarrow{y}.\, (\overrightarrow{x}.\widehat{\mathrm{S}}) \diamond \mathsf{cy}^m(\overrightarrow{x}.\, (\overrightarrow{y}.\widehat{\mathrm{T}}) \diamond \mathsf{cy}^n(\overrightarrow{y}.\widehat{\mathrm{S}})))\rangle$$

Cleaning rules for c satisfying $\mathsf{AxBr}([\,], +)$

$$(11\mathrm{r})\quad \mathsf{cy}(x^{\mathsf{Var}_c}.\mathsf{v}(x) + \mathrm{T}) \quad \to \mathrm{T}$$

$$(12\mathrm{r})\quad \mathsf{cy}(x^{\mathsf{Var}_c}.\mathrm{T} + \mathsf{v}(x)) \quad \to \mathrm{T}$$

$$(13\mathrm{r})\quad \mathsf{cy}(\overrightarrow{y}.\mathrm{T}) \quad \to \mathrm{T}$$

$$(14\mathrm{r})\quad \mathsf{cy}(x^{\mathsf{Var}_c}.\mathsf{v}(x)) \quad \to [\,]$$

$$(15\mathrm{r})\quad [\,] + \mathrm{T} \quad \to \mathrm{T}$$

$$(16\mathrm{r})\quad \mathrm{T} + [\,] \quad \to \mathrm{T}$$

In (7r), $|\overrightarrow{\mathrm{S}}| = 1$ is also allowed and the part $\langle\,\overrightarrow{\mathrm{S}}\,\rangle$ is simply a single metavariable S.
By the notation of metavariable, in (13r), $\mathsf{cy}(\overrightarrow{y}.\mathrm{T})$ means that T cannot involve $\overrightarrow{y}$, and similar for (12r)(13r). The rules (11r)-(16r) are parameterised by the function symbols $[\,], +$ as in $\mathsf{AxBr}([\,], +)$.

Figure 8: Second-order rewrite system SIMP for simplification

We omit writing contexts and types of equations for simplicity. For example, formally the axiom (1) in FOLD is written as

$$\mathrm{E}_1 : \overrightarrow{a_1} \to c, \ldots, \mathrm{E}_n : \overrightarrow{a_1} \to c \triangleright \ \vdash \ \mathsf{fold}^c_b(\mathrm{E}_1, \ldots, \mathrm{E}_n,\ \overrightarrow{y}.\, y_i\,;\ \overrightarrow{y}) \ = \ y_i \ : \ b$$

The arguments of fold expressing the c-structure are abbreviated as E for simplicity.

We state the correctness of this axiomatisation, which holds by the above faithful construction. The following is immediate by construction.

Proposition 4.1. The following are equivalent.

- $\mathbf{fold}^c_b(\alpha)([\Gamma \vdash t \ : c]_{\mathcal{E}}) = [\Gamma' \vdash u \ : b]_{\mathcal{E}}$
- $\vdash \mathsf{fold}^c_b(\overrightarrow{x_1}.e_1, \ldots, \overrightarrow{x_m}.e_m,\ \overrightarrow{y}.\,t\,;\ \overrightarrow{y}) = u \ : b$ is derived from the axioms $\mathcal{E} \cup \mathsf{FOLD}$ using the second-order equational logic.

where α, e_i and t are fold free, $\Gamma = y_1 : c, \ldots, y_n : c,\ \Gamma' = y_1 : [\![c]\!], \ldots, y_n : [\![c]\!]$.

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
```

We specify `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 defined by fold where the target `CNat`-structure is defined to be $[\![\mathtt{0}]\!]^M = n, [\![\mathtt{S}]\!]^M(x) = \mathtt{S}(x)$.

4.2. **Primitive recursion by fold.** The fold axiomatised 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 [MFP91] 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.

4.3. **Assumptions.** Let $(c, \Sigma_c, \mathcal{E}_c)$ and $(b, \Sigma_b, \mathcal{E}_b)$ be datatype declarations, where $\Sigma_c = \{d_1, \ldots, d_n\}$. We suppose the following form of specification and definition.

$$\begin{array}{lrcl} f & : c & \to & b \\ \mathtt{spec} & f(d_1(\overrightarrow{a}, \overrightarrow{t})) & = & e_{d_1} \\ & & \cdots & \\ & f(d_n(\overrightarrow{a}, \overrightarrow{t})) & = & e_{d_n} \\ & & & \\ \mathtt{fun} & \multicolumn{3}{l}{f(t) = \pi_1 \diamond \mathsf{fold}^c_{b,c}(\,(v_1, w_1, \ldots, v_n, w_n.\langle e'_d,\ d(\overrightarrow{a}, \overrightarrow{w})\rangle)_{d \in \Sigma_c},\ t\,)} \end{array} \tag{4.3}$$

We define $\pi_1 \triangleq (v, w.\ v)$. We assume that every e_d is a closed term of type b, which may involve terms of the types

$$\vdash f(t_i) \ : b \qquad \vdash t_i \ : c$$

for each $i = 1, \ldots, n$, (i.e., $\overrightarrow{t} = t_1, \ldots, t_n$ may appear solely, cf. examples in §7 in e_d). We define e'_d to be a term obtained from e_d, by replacing every $f(t_i)$ with v_i of type b, and every t_i (not in the form $f(t_i)$) with w_i of type c, and assume

$$v_1 : b, w_1 : c, \ldots, v_n : b, w_n : c \vdash e'_d \ : b.$$

The above specification can be seen as describing primitive recursion, because it is similar to 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 [Mee92]. We now take the fold where the target Σ-structure is the sequence b, c of types, i.e., $\mathsf{fold}^c_{b,c}$.

4.4. **Structure.** Let $\mathcal{E}$ be the set of axioms collecting the axioms of all datatypes in $\mathcal{B}$. Hence $\mathcal{E} \supseteq \mathcal{E}_b \cup \mathcal{E}_c$. We define a $(\Sigma_c, \mathcal{E})$-structure (M, α) in $\mathbf{Tm}(\mathcal{E})$ by

$$[\![c]\!]^M = (b, c)$$

and $[\![a]\!]^M = a$, otherwise. If $\mathcal{E}_c$ contains axioms $\mathsf{AxBr}([\,], +)$, then $\mathcal{E}_b$ must contains axioms $\mathsf{AxBr}(\nu, \mu)$, where $\nu : b,\ \mu : b, b \to b$ are in Σ_b. We define

$$(\![\,[\,]\,]\!)^M \ = \ \langle \eta, [\,] \rangle, \qquad (\![+]\!)^M(\langle v_1, w_1\rangle, \langle v_2, w_2\rangle) \ = \ \langle \mu(v_1, v_2), w_1 + w_2 \rangle$$
$$(\![d]\!)^M(\overrightarrow{a}, v_1, w_1, \ldots, v_n, w_n) \ = \ \langle e'_d,\ d(\overrightarrow{a}, \overrightarrow{w}) \rangle$$

for $d : \overrightarrow{a}, c^n \to c \in \Sigma_{\mathsf{con}}$.

4.5. **Formalisation.** Then we have the function

$$\mathbf{fold}^c_{b,c}(\alpha) : \mathbf{Tm}(\mathcal{E})(\overrightarrow{c},\ c) \longrightarrow \mathbf{Tm}(\mathcal{E})([\![\overrightarrow{c}]\!],\ (b, c)) \tag{4.4}$$

which is axiomatised as the function symbol

$$\mathsf{fold}^c_{b,c} : ([\![\overrightarrow{a_1}]\!] \to (b, c)),\ \ldots, ([\![\overrightarrow{a_m}]\!] \to (b, c)), (\overrightarrow{c} \to c), [\![\overrightarrow{c}]\!] \longrightarrow (b, c).$$

The axioms FOLD in Fig. 6 is instantiated to the case of $\mathsf{fold}^c_{b,c}$. For example,

$$(1) \qquad \mathsf{fold}^c_{b,c}(E,\ \overrightarrow{x}.\, x_i\ ;\ v_1, w_1, \ldots, v_n, w_n) \ = \ \langle v_i, w_i \rangle$$

Other axioms for fold are instantiated similarly.

4.6. **Properties.** By construction,

$$\vdash e_d \ = \ (v_1, w_1, \ldots, v_n, w_n.\, e'_d) \diamond \langle f(t_1), t_1, \ldots, f(t_n), t_n \rangle \ \ : b$$

holds. Define $E \triangleq (v_1, w_1, \ldots, v_n, w_n.e'_d)_{d \in \Sigma_c}$. By induction on the typing derivations, we have

$$\mathsf{fold}^c_{b,c}(E,\ t) = \langle\, f(t),\ t \,\rangle$$

for all closed terms t of type c. By the characterisation (3.2), we have for each $d \in \Sigma_{\mathsf{con}}$,

$$\begin{aligned}\langle f(d(\overrightarrow{a}, \overrightarrow{t})), t\rangle &= \mathsf{fold}^c_{b,c}(E,\ f(d(\overrightarrow{a}, \overrightarrow{t}))) \\ &= \langle (v_1, w_1, \ldots, v_n, w_n.\langle e'_d, d(\overrightarrow{a}, \overrightarrow{w})\rangle) \diamond \langle\, f(t_1), t_1, \ldots, f(t_n), t_n \,\rangle,\ t\rangle = \langle e_d,\ t\rangle\end{aligned}$$

hence we have

$$f(d(\overrightarrow{a}, \overrightarrow{t})) = e_d$$

meaning that f satisfies the specification. We use this representation of primitive recursion in §7.

Remark 4.3. If e_d constrains only recursive calls of the forms $f(t_i)$, it is merely a pattern of structural recursion, so it can be implemented by fold using the structure $v.e'$ where all the recursive calls $f(t_i)$ in e_d are abstracted to v as Example 4.2.

5. Strongly Normalising Computation Rules for FOLD

We expect that FOLD provides strongly 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 FOLD that is strictly decreasing is difficult for this case. Such a naive method is typically dangerous for higher-order rewrite rules and might lead one to an unintentional mistake. If the axioms regarded as second-order rules are a binding Combinatory Reduction System (CRS) [Ham05] (cf. Remark 2.2), meaning that every meta-application $\text{M}[t_1, \dots, t_n]$ is of the form $\text{M}[\overrightarrow{x}]$, then it is possible to use a simple polynomial interpretation to prove termination of second-order rules [Ham05]. Unfortunately, this is not the case because in (5) there is a meta-application 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* [BJO02, Bla00], which is based on Tait's computability method to show *strong normalisation* (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 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" [Coq92], but more relaxed and extended.

5.1. **General Schema.** We review the General Schema criterion [Bla00, Bla16]. For more details and the proofs, see the original papers [Bla00, Bla16].

The General Schema is formulated for a framework of rewrite rules called inductive datatype systems, whose second-order fragment is essentially the same as the present formulation given in §2. Minor differences are adapted as follows. We use greek letters such as $\alpha, \sigma, \tau, \dots$ to denote types. Roman alphabets such as $a, b, c, \dots$ are used for base types.

(i) The target of function symbols must be a single (not necessary base) type in an inductive datatype system. Hence we introduce the product type constructor $\times$, assume that $b_1 \times b_2$ is again a base type in the sense of §2.3, and use it for the target type.

(ii) Instead of a term $x_1, \dots, x_n.t$ of sort $a_1, \dots, a_n \to b$ in a second-order algebraic theory, we use $x_1. \cdots .x_n.t$ of type $a_1 \to \cdots \to a_n \to b$. Now the abbreviation $\overrightarrow{x}.t$ denotes $x_1. \cdots .x_n.t$.

Definition 5.1. A set of rewrite rules induces the following relation on function symbols in a signature Σ: f *depends on* g if there is a rewrite rule defining f (i.e., whose left-hand side is headed by f) in the right-hand side of which g occurs. Its transitive closure is denoted by $>_\Sigma$.

Definition 5.2. A *constructor* is a function symbol $c : \overrightarrow{\tau} \to b$ which does not occur at the root symbol of the left-hand side of any rule. The set of all constructors defines a preorder $\leq_\mathcal{B}$ on the set $\mathcal{B}$ of mol types by $a \leq_\mathcal{B} b$ if b occurs in $\overrightarrow{\tau}$ for a constructor $c : \overrightarrow{\tau} \to b$. Let $<_\mathcal{B}$ be the strict part and $=_\mathcal{B}$ the equivalence relation generated by $\leq_\mathcal{B}$. A type b is *positive* if for any type a s.t. $a =_\mathcal{B} b$, a does not occur at a negative position of the type of constructor c of b. A constructor $c : \overrightarrow{\tau} \to b$ is positive if b is positive.

Definition 5.3. A metavariable Z is *accessible* in a meta-term t if there are distinct bound variables $\overrightarrow{x}$ such that $\mathrm{Z}[\overrightarrow{x}] \in \mathsf{Acc}(t)$, where $\mathsf{Acc}(t)$ is the least set satisfying the following clauses:

(a1) $t \in \mathsf{Acc}(t)$.

(a2) If $x.u \in \mathsf{Acc}(t)$ then $u \in \mathsf{Acc}(t)$.

(a3) If $c(\overrightarrow{s}) \in \mathsf{Acc}(t)$ then each $s_i \in \mathsf{Acc}(t)$ for a constructor c.

(a4) Let u_i be a term of type τ_i for each $i = 1, \ldots, n$. If $\mathsf{Acc}(t) \ni f(u_1, \ldots, u_n)$ is of a base type b, and b (possibly) occurs positively in type τ_i, then $u_i \in \mathsf{Acc}(t)$ [Bla16, Def.15].

A position p is a finite sequence of natural numbers. The order on positions is defined by $p < q$ if there exists a non-empty p' such that $p\,p' = q$. Given a meta-term t, $\mathcal{P}os(t)$ denotes the set of all positions of t. The notation $t_{|p}$ denotes a subterm of t at a position p, and $t[s]_p$ the term obtained from t by replacing the subterm at the position p by s.

Definition 5.4. A meta-term u is a *covered-subterm* of t, written $t \mathrel{\widehat{\trianglerighteq}} u$, if there are two positions $p \in \mathcal{P}os(t), q \in \mathcal{P}os(t_{|p})$ such that

- $u = t[t_{|pq}]_p$,
- $\forall r < p.\ t_{|r}$ is headed by an abstraction, and
- $\forall r < q.\ t_{|pr}$ is headed by a function symbol, including a constructor.

For example, $f(a, c(x)) \mathrel{\widehat{\trianglerighteq}} c(x)$, and $\lambda(x.\mathrm{M}[x]) \mathrel{\widehat{\trianglerighteq}} \mathrm{M}[x]$, and $y.\lambda(x.\mathrm{M}[x]) \mathrel{\widehat{\trianglerighteq}} y.x.\mathrm{M}[x]$.

Definition 5.5. Given $f : \tau_1, \ldots, \tau_n \to \tau \in \Sigma$, the *computable closure* $\mathcal{CC}_f(\overrightarrow{t})$ of a meta-term $f(\overrightarrow{t})$ is the least set $\mathcal{CC}$ satisfying the following clauses. We assume that all the terms below are well-typed.

(1) If $\mathrm{Z} : \tau_1, \ldots, \tau_p \to \tau$ is accessible in some of $\overrightarrow{t}$, and $\overrightarrow{u} \in \mathcal{CC}$, then $\mathrm{Z}[\overrightarrow{u}] \in \mathcal{CC}$.

(2) For any variable x, $x \in \mathcal{CC}$.

(3) If $c : \tau_1, \ldots, \tau_n \to b$ is a constructor and $\overrightarrow{u} \in \mathcal{CC}$, then $c(\overrightarrow{u}) \in \mathcal{CC}$.

(4) If $u, v \in \mathcal{CC}$, then $u@v \in \mathcal{CC}$ for $@ : (\sigma \to \tau), \sigma \to \tau$.

(5) If $u \in \mathcal{CC}$ then $x.u \in \mathcal{CC}$.

(6) If $f >_\Sigma g$ and $\overrightarrow{w} \in \mathcal{CC}$, then $g(\overrightarrow{w}) \in \mathcal{CC}$.

(7) If $\overrightarrow{u} \in \mathcal{CC}$ such that $\overrightarrow{t} \mathrel{\widehat{\rhd}}^{\mathsf{lex}} \overrightarrow{u}$, then $f(\overrightarrow{u}) \in \mathcal{CC}$, where $\widehat{\rhd}^{\mathsf{lex}}$ is the lexicographic extension of the strict part of $\widehat{\trianglerighteq}$.

Definition 5.6. A rewrite rule $f(\overrightarrow{t}) \to r$ *satisfies* the General Schema if $\mathcal{CC}_f(\overrightarrow{t}) \ni r$.

Theorem 5.7. ([Bla00]) Suppose that given a signature Σ and rules $\mathcal{R}$ satisfies the following:

(1) $>_{\mathcal{B}}$ is well-founded,
(2) every constructor is positive, and
(3) $>_{\Sigma}$ is well-founded.

If all the rules of $\mathcal{R}$ satisfy the General Schema, then $\mathcal{R}$ is strongly normalising.

5.2. **Refining second-order algebraic theory to second-order rewrite rules.** In order to apply the General Schema criterion, we refine the second-order algebraic theories AxCy, AxBr and FOLD to the second-order rewrite rules FOLDr and SIMP.

Crucially, the constructors used in FOLD are not positive, as cy and $\diamond$ involve a 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 the type of "variables" of type c. We assume the constructor v which embeds a "variable" into a term. We modify the types of default constructors as follows:

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

where c's and a's are base types of the inductive datatype system (which may be product types), $\overrightarrow{\mathsf{Var}_a} \to c$ is short for $\mathsf{Var}_{a_1} \to \cdots \to \mathsf{Var}_{a_n} \to c$. The type of fold is now

$$\mathsf{fold}^c_b : (\overrightarrow{\mathsf{Var}_{a_1}} \to b), \ldots, (\overrightarrow{\mathsf{Var}_{a_k}} \to b), (\mathsf{Var}^m_c \to c), \mathsf{Var}^m_b \longrightarrow b,$$

The use of a type $\mathsf{Var}_\sigma \to \tau$ to represent binders is well-known in the field of mechanized reasoning, sometimes called *(weak) higher-order abstract syntax* [DFH95].

We also modifty the type of composition as

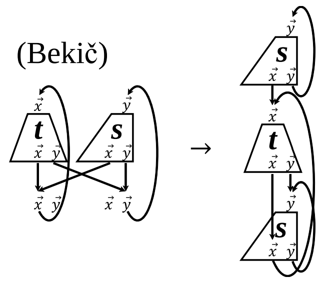
$$- \diamond - : (a_1, \ldots, a_n \to c), a_1 \times \cdots \times a_n \to c$$

which is not a constructor in the sense of Def. 5.2, hence it is allowed as not positive (when $a_i = c$).

5.3. **Second-order rewrite systems FOLDr and SIMP.** We now describe how we obtain the second-order rewrite rules FOLDr and SIMP given in Fig. 7 and 8. Note that FOLDr's "r" stands for "rewrite". The rewrite system FOLDr is an oriented version of FOLD. The axiom (5) is refined to (5r) by applying **Composition** rules which is need to prove SN. Because of it, a rule corresponding to (6) is not needed in FOLDr. The axiom (sub) in AxCy involves a meta-application at the right-hand side, which performs general substitution of terms for variables $\overrightarrow{y}$ of base types. The oriented version of it is the rule (7r).

The rewrite system SIMP is for simplification, whose rules are taken from the equational theory of $\mathsf{AxCy} \cup \mathsf{AxBr}$. We include the Bekič law as a rewrite rule (10r), which can be depicted as:

(Bekič)



It says that the fixed point of a pair can be obtained by computing the fixed points 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. The superscript of cy in (10r) indicates the length of the tuple argument (see §2.1). Such a superscript indicating an invariant of the arguments is similar to the idea of higher-order semantic labelling [Ham07], but here we just make the existing superscript explicit rather than labelling. Hence it is suitable for rewrite rules and actually shown to be terminating (cf. the proof of Thm. 5.9).

We define the relation $\to_{\mathsf{FOLDr}}$ (resp. $\to_{\mathsf{SIMP}}$) on terms by the relation generated by the second-order equational logic Fig. 3 under the axioms FOLDr (resp. SIMP) without using the (Ax2), (Ref) and (Tra)-rules. Namely, "one-step rewriting" $\to_{\mathsf{FOLDr}}$ is equational reasoning without using symmetry, reflexivity and transitivity. By construction, we immediately see that the rewrite system FOLDr correctly implements the second-order algebraic theory FOLD. The following proposition is immediate by construction of rules. We write $\check{t}$ for a term that recovers the original terms t by stripping the constructor v.

Proposition 5.8. If $t \to^{+}_{\mathsf{FOLDr}} t'$, then $\check{t} = \check{t}'$ is derivable from $\mathsf{FOLD} \cup \mathsf{AxCy} \cup \mathsf{AxBr}$.

Theorem 5.9. The second-order rewrite system $\mathsf{FOLDr} \cup \mathsf{SIMP}$ is strongly normalising.

Proof. We use the the General Schema criterion using the well-founded relation $>_\Sigma$ on function symbols

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

where natural numbers $m > n \geq 1$. Now all constructors are positive.

In the following proof, we write $f(\overrightarrow{t}) \succ s$ when $\mathcal{CC}_f(\overrightarrow{t}) \ni s$. We show that $\mathsf{FOLDr} \cup \mathsf{SIMP}$ satisfies the General Schema, i.e., for each rewrite rule $f(\overrightarrow{t}) \to s$, we check $f(\overrightarrow{t}) \succ s$.

(1r) $\mathsf{fold}(E, \overrightarrow{y}.\mathsf{v}(y_i); \overrightarrow{y}) \succ \mathsf{v}(y_i) \overset{(6)}{\Longleftarrow} \mathsf{fold}(E, \overrightarrow{y}.\mathsf{v}(y_i); \overrightarrow{y}) \succ y_i$, which holds by (2).

The implication "$\Longleftarrow$" expresses backward inference, which is labelled with a number indicating which clause in Def. 5.3 or Def. 5.5 is applied. In what follows, for brevity, we examine mostly the cases that the length of bound variables is 1 (i.e., $|\overrightarrow{x}| = |\overrightarrow{y}| = 1$) and the arity of d is 2. General cases are proved similarly.

(4r) $\mathsf{fold}(y.\mathsf{cy}(x.\mathrm{T}[y,x]); y) \succ \mathsf{cy}(x.\mathsf{fold}(y, x.\mathrm{T}[y,x]; y))$

$\overset{(6)}{\Longleftarrow}$ $\mathsf{fold}(y.\mathsf{cy}(x.\mathrm{T}[y,x]); y) \succ \mathsf{fold}(y, x.\mathrm{T}[y,x]; y)$

$\overset{(7)}{\Longleftarrow}$ $\mathsf{fold}(y.\mathsf{cy}(x.\mathrm{T}[y,x]); y) \succ y$

& $\mathsf{fold}(y.\mathsf{cy}(x.\mathrm{T}[y,x]); y) \succ y, x.\mathrm{T}[y,x]$ & $y.\mathsf{cy}(x.\mathrm{T}[y,x]) \mathrel{\widehat{\trianglerighteq}} y, x.\mathrm{T}[y,x]$

The second literal holds easily. In general, if a metavariable with binders is a subterm of a larger meta-term, it is proved to be smaller by $\succ$. The third literal of a covered subterm relation holds, because stripping the prefix binders, it is a subterm relation.

(5r) $\mathsf{fold}(z, x.\mathrm{E}_d[z,x], y.d(\mathrm{A}, \mathrm{T}[y]); y) \succ \mathrm{E}_d[\mathrm{A}, \mathsf{fold}(z, x.\mathrm{E}_d[z,x], y.\mathrm{T}[y]; y)]$
$\stackrel{(6)}{\Longleftarrow}$ $\mathsf{fold}(z, x.\mathrm{E}_d[z,x], y.d(\mathrm{A}, \mathrm{T}[y]); y) \succ \mathrm{A},\ \mathsf{fold}(z, x.\mathrm{E}_d[z,x], y.\mathrm{T}[y]; y)$
& E_d is accessible in $(z, x.\mathrm{E}_d[z,x])$.

The first literal is proved to be smaller straightforwardly (cf. (4r)) and the accessiblity $\mathsf{Acc}(z, x.\mathrm{E}_d[z,x]) \ni \mathrm{E}_d[z,x]$ is shown by (a1)(a2). The general case that the part $x.\mathrm{E}_d[x]$ in fold is a sequence of metavariables $(x.\mathrm{E}_d[x])_{d\in\Sigma_c}$ is similar. Hereafter, we omit writing the part $E = (x.\mathrm{E}_d[x])_{d\in\Sigma_c}$ and the arguments after ";" in fold for brevity.

(3r) $\mathsf{fold}(y.\langle \mathrm{S}[y], \mathrm{T}[y]\rangle) \succ \langle \mathsf{fold}(y.\mathrm{S}[y]), \mathsf{fold}(y.\mathrm{T}[y])\rangle$
$\stackrel{(3)}{\Longleftarrow}$ $\mathsf{fold}(y.\langle \mathrm{S}[y], \mathrm{T}[y]\rangle) \succ \mathsf{fold}(y.\mathrm{S}[y])$ & $\mathsf{fold}(y.\langle \mathrm{S}[y], \mathrm{T}[y]\rangle) \succ \mathsf{fold}(y.\mathrm{T}[y])$
$\stackrel{(7)}{\Longleftarrow}$ $y.\langle \mathrm{S}[y], \mathrm{T}[y]\rangle \mathrel{\widehat{\trianglerighteq}} y.\mathrm{S}[y]$ & $y.\langle \mathrm{S}[y], \mathrm{T}[y]\rangle \mathrel{\widehat{\trianglerighteq}} y.\mathrm{T}[y]$

(7r) $(\overrightarrow{y}.\,\mathrm{T}[\overrightarrow{y}]) \diamond \langle \overrightarrow{\mathrm{S}} \rangle \succ \mathrm{T}[\overrightarrow{\mathrm{S}}]$
$\stackrel{(1)}{\Longleftarrow}$ $(\overrightarrow{y}.\,\mathrm{T}[\overrightarrow{y}]) \diamond \langle \overrightarrow{\mathrm{S}} \rangle \succ \overrightarrow{\mathrm{S}}$ & $\mathsf{Acc}(\overrightarrow{y}.\mathrm{T}[\overrightarrow{y}]) \ni \mathrm{T}[\overrightarrow{y}]$

The first literal holds because $\overrightarrow{\mathrm{S}}$ are accesible in $\langle \overrightarrow{\mathrm{S}} \rangle$. The second literal holds by (a2)(a1).

(10r) We write $\widehat{\mathrm{T}} = \mathrm{T}[\overrightarrow{x}, \overrightarrow{y}], \widehat{\mathrm{S}} = \mathrm{S}[\overrightarrow{x}, \overrightarrow{y}]$.
$\mathsf{cy}^{m+n}(\overrightarrow{x}, \overrightarrow{y}.\langle \widehat{\mathrm{T}}, \widehat{\mathrm{S}}\rangle) \succ \langle \mathsf{cy}^m(\overrightarrow{x}.(\overrightarrow{y}.\widehat{\mathrm{T}}) \diamond \mathsf{cy}^n(\overrightarrow{y}.\widehat{\mathrm{S}})), \mathsf{cy}^n(\overrightarrow{y}.(\overrightarrow{x}.\widehat{\mathrm{S}}) \diamond \mathsf{cy}^m(\overrightarrow{x}.(\overrightarrow{y}.\widehat{\mathrm{T}}) \diamond \mathsf{cy}^n(\overrightarrow{y}.\widehat{\mathrm{S}}))))\rangle$
$\stackrel{(6)}{\Longleftarrow}$ $\mathsf{cy}^{m+n}(\overrightarrow{x}, \overrightarrow{y}.\langle \widehat{\mathrm{T}}, \widehat{\mathrm{S}}\rangle) \succ \mathsf{cy}(\overrightarrow{x}.(\overrightarrow{y}.\widehat{\mathrm{T}}) \diamond \mathsf{cy}(\overrightarrow{y}.\widehat{\mathrm{S}}))$
& $\mathsf{cy}^{m+n}(\overrightarrow{x}, \overrightarrow{y}.\langle \widehat{\mathrm{T}}, \widehat{\mathrm{S}}\rangle) \succ \mathsf{cy}(\overrightarrow{y}.(\overrightarrow{x}.\widehat{\mathrm{S}}) \diamond \mathsf{cy}(\overrightarrow{x}.(\overrightarrow{y}.\widehat{\mathrm{T}}) \diamond \mathsf{cy}(\overrightarrow{y}.\widehat{\mathrm{S}})))$ $\triangleq$ (A) & (B).
In what follows, unlabelled cy denotes cy^m or cy^n.

(A) $\mathsf{cy}^{m+n}(\overrightarrow{x}, \overrightarrow{y}.\langle \widehat{\mathrm{T}}, \widehat{\mathrm{S}}\rangle) \succ \mathsf{cy}(\overrightarrow{x}.(\overrightarrow{y}.\widehat{\mathrm{T}}) \diamond \mathsf{cy}(\overrightarrow{y}.\widehat{\mathrm{S}}))$ $\stackrel{(6)}{\Longleftarrow}$ $\mathsf{cy}^{m+n}(\overrightarrow{x}, \overrightarrow{y}.\langle \widehat{\mathrm{T}}, \widehat{\mathrm{S}}\rangle) \succ \overrightarrow{x}.(\overrightarrow{y}.\widehat{\mathrm{T}}) \diamond \mathsf{cy}(\overrightarrow{y}.\widehat{\mathrm{S}})$ $\stackrel{(6)}{\Longleftarrow}$ $\mathsf{cy}^{m+n}(\overrightarrow{x}, \overrightarrow{y}.\langle \widehat{\mathrm{T}}, \widehat{\mathrm{S}}\rangle) \succ \overrightarrow{x}.(\overrightarrow{y}.\widehat{\mathrm{T}}),\ \mathsf{cy}(\overrightarrow{y}.\widehat{\mathrm{S}})$

(B) $\mathsf{cy}^{m+n}(\overrightarrow{x}, \overrightarrow{y}.\langle \widehat{\mathrm{T}}, \widehat{\mathrm{S}}\rangle) \succ \mathsf{cy}(\overrightarrow{y}.(\overrightarrow{x}.\widehat{\mathrm{S}}) \diamond \mathsf{cy}(\overrightarrow{x}.(\overrightarrow{y}.\widehat{\mathrm{T}}) \diamond \mathsf{cy}(\overrightarrow{y}.\widehat{\mathrm{S}})))$
$\stackrel{(6)}{\Longleftarrow}$ $\mathsf{cy}^{m+n}(\overrightarrow{x}, \overrightarrow{y}.\langle \widehat{\mathrm{T}}, \widehat{\mathrm{S}}\rangle) \succ \overrightarrow{y}.(\overrightarrow{x}.\widehat{\mathrm{S}}) \diamond \mathsf{cy}(\overrightarrow{x}.(\overrightarrow{y}.\widehat{\mathrm{T}}) \diamond \mathsf{cy}(\overrightarrow{y}.\widehat{\mathrm{S}}))$
$\stackrel{(6)}{\Longleftarrow}$ $\mathsf{cy}^{m+n}(\overrightarrow{x}, \overrightarrow{y}.\langle \widehat{\mathrm{T}}, \widehat{\mathrm{S}}\rangle) \succ \overrightarrow{y}.(\overrightarrow{x}.\widehat{\mathrm{S}})$ & $\mathsf{cy}^{m+n}(\overrightarrow{x}, \overrightarrow{y}.\langle \widehat{\mathrm{T}}, \widehat{\mathrm{S}}\rangle) \succ \mathsf{cy}(\overrightarrow{x}.(\overrightarrow{y}.\widehat{\mathrm{T}}) \diamond \mathsf{cy}(\overrightarrow{y}.\widehat{\mathrm{S}}))$

The first literal easily holds. The second literal holds by

$\stackrel{(6)}{\Longleftarrow}$ $\mathsf{cy}^{m+n}(\overrightarrow{x}, \overrightarrow{y}.\langle \widehat{\mathrm{T}}, \widehat{\mathrm{S}}\rangle) \succ \overrightarrow{x}.(\overrightarrow{y}.\widehat{\mathrm{T}}) \diamond \mathsf{cy}(\overrightarrow{y}.\widehat{\mathrm{S}})$
$\stackrel{(6)}{\Longleftarrow}$ $\mathsf{cy}^{m+n}(\overrightarrow{x}, \overrightarrow{y}.\langle \widehat{\mathrm{T}}, \widehat{\mathrm{S}}\rangle) \succ \overrightarrow{x}.(\overrightarrow{y}.\widehat{\mathrm{T}}),$ $\mathsf{cy}(\overrightarrow{y}.\widehat{\mathrm{S}}),$ each of which holds easily.

The remaining cases are simpler and similarly proved. By Thm. 5.7, we have strong normalisation of $\mathsf{FOLDr} \cup \mathsf{SIMP}$. $\square$

Remark 5.10. This strong normalisation result is very general. Not only ensuring termination of computation of $\mathsf{fold}^c_b(\overrightarrow{x_1}.e_1,\ldots,\overrightarrow{x_m}.e_m,\ t\,)$ for *closed terms* $\overrightarrow{e},t$ (which is an expected result by the semantics characterisation), the result ensures that *any term* $\mathsf{fold}^c_b(\overrightarrow{x_1}.e_1,\ldots,\overrightarrow{x_m}.e_m,\ \overrightarrow{y}.\,t\,;\ \overrightarrow{y})$ involving possibly

(i) multiple fold's, or even nested fold's and

(ii) free variables (i.e., e and t can be *open terms*)

is strongly normalising without imposing any reduction strategy. For (i), consider the situation that one defines a function which calls other functions. Since in this paper, our methodology is that any function on cyclic datatypes is defined using fold, this situation is realised using fold involving other fold's.

6. Decidability of Equational Theory

In this section, we show an important property of our framework, namely, the decidability of the equational theory generated by AxCy, AxBr and FOLDr. This is done by investigating another important rewriting property, *Church-Rosser modulo bisimulation* of FOLDr. SN of FOLDr established in the previous section also plays an important role to establish Church-Rosser modulo bisimulation. Note that since the rewrite rules SIMP is merely a subset of an oriented version of theorems derived from $\mathsf{AxCy} \cup \mathsf{AxBr}$, we do not need to include SIMP for the decidability of equational theory.

Notation 6.1. Hereafter, we omit writing the variable term constructor v in terms used in FOLDr for simplicity. For example, we will simply write $\mathsf{cy}(x.\,x)$ to mean $\mathsf{cy}(x.\,\mathsf{v}(x))$. We may write simply $\to$ for the rewrite relation $\to_{\mathsf{FOLDr}}$. We write $\to^*$ for the reflexive transitive closure, $\to^+$ for the transitive closure, and $\leftarrow$ for the converse of $\to$. We define $\leftrightarrow \triangleq \to \cup \leftarrow$. The notation $\mathbf{nf}(t)$ denotes a unique normal form of t. We write $t \overset{!}{\to} t'$ if $t \to^* t'$ and t' is a normal form, meaning rewriting to a normal form.

6.1. Church-Rosser modulo $\sim$.

An important property for rewriting with equational theory is Church-Rosser modulo equivalence relation [Hue80, Ter03].

A relation $\to$ is *Church-Rosser modulo* $\sim$ (CR$_\sim$) if $s\ (\sim \cup \leftrightarrow)^*\ t$ implies there exist s', t' such that $s \to^* s'$ & $t \to^* t'$ & $s' \sim t'$. In diagram,

$$\begin{array}{ccc} s & (\sim \cup \leftrightarrow)^* & t \\ {\scriptstyle *}\downarrow & & \downarrow{\scriptstyle *} \\ s' & \sim & t' \end{array} \tag{6.1}$$

Definition 6.2. We define $\sim$ to be the equivalence relation on terms generated by $\mathsf{AxCy} \cup \mathsf{AxBr}$, i.e., bisimulation.

6.2. **Confluence on plain terms.** We consider confluence on terms without any quotient.

Proposition 6.3. The relation $\to_{\mathsf{FOLDr}}$ is confluent.

Proof. Since the only critical pair between (6) and (7r) is joinable, $\to_{\mathsf{FOLDr}}$ is locally confluent. $\to_{\mathsf{FOLDr}}$ is also SN (Thm. 5.9). By Newman's lemma [Hue80, BN98], it is confluent [Klo80]. $\square$

Note that this result does not contradict known counterexamples to confluence of graph rewriting systems [AK96],[Has97, Example 2.4.3]. Counterexamples to confluence in [AK96] use the fixed point law as a rewrite rule, while our FOLDr does not have the fixed point law.

6.3. **Normal forms.**

Proposition 6.4. The normal form of a term by rewriting with FOLDr is unique.

Proof. Since FOLDr is confluent. $\square$

We analyse the structure of normal forms. We call a term a *value* if it follows the grammar ($d \in \Sigma_{\mathsf{con}}$):

$$t, t_1, \ldots, t_n ::= y \mid d(\overrightarrow{t}) \mid \mathsf{cy}(\overrightarrow{x}.t) \mid \langle t_1, t_2 \rangle \mid \langle\,\rangle \mid (\overrightarrow{y}.t_1) \diamond t_2$$

Proposition 6.5. Suppose $\overrightarrow{x_1} \vdash e_1 \ : c, \ldots, \overrightarrow{x_m} \vdash e_m \ : c$ and define $e \triangleq \overrightarrow{x_1}.e_1, \ldots, \overrightarrow{x_m}.e_m$. If $\overrightarrow{y : b} \vdash t \ : \overrightarrow{c}$ is a value, then $\mathbf{nf}(\ \mathsf{fold}(e, \overrightarrow{y}.t\,;\ \overrightarrow{y})\ )$ is a value.

Proof. We abbreviate $\mathsf{fold}(e, \overrightarrow{y}.t; \overrightarrow{y})$ as $\mathsf{fold}(t)$. By induction on the structure of values. Base cases: y and $\langle\,\rangle$'s normal forms are themselves. Induction step: Case $t = \langle t_1, t_2 \rangle$ is a value.

$$\mathbf{nf}(\mathsf{fold}(\langle t_1, t_2 \rangle)) = \mathbf{nf}(\langle \mathsf{fold}(t_1), \mathsf{fold}(t_2) \rangle) = \langle \mathbf{nf}(\mathsf{fold}(t_1)), \mathbf{nf}(\mathsf{fold}(t_2)) \rangle \overset{\text{I.H.}}{=} \langle v_1, v_2 \rangle$$

where v_1, v_2 are values. The cases $d(\overrightarrow{t}), \mathsf{cy}(x.t), (\overrightarrow{y}.t_1) \diamond t_2$ are similar. $\square$

The propositions Prop. 4.1 and Prop. 6.5 show that FOLD is a correct implementation of the characterisation (3.2) stating that fold preserves bisimilarity and is total.

6.4. **A decidable proof method for the equational theory.** $\mathrm{CR}_\sim$ is a desirable property, because the diagram (6.1) describes a proof method of $s\ (\sim \cup \leftrightarrow)^*\ t$, i.e., $s = t$ is derivable from $\mathsf{AxCy} \cup \mathsf{AxBr}$ and the equational theory generated form FOLDr.

We explain the reason why $\mathrm{CR}_\sim$ is useful below. We suppose that $\mathrm{CR}_\sim$ holds. Now $\mathrm{CR}_\sim$ means that a proof of

$$s\ (\sim \cup \leftrightarrow)^*\ t,$$

where s and t are connected by disordered combinations of $\sim$, $\to$, $\leftarrow$, is always transformed to a proof

$$s\ \to^*\ s'\ \sim\ t'\ \leftarrow^*\ t$$

for some s', t'. But this is still not good enough, because it is not clear how many times we should rewrite s and t to reach suitable s' and t'. For the case of FOLDr, we can further transform it to an equivalent proof (cf. Notation 6.1)

$$s\ \overset{!}{\to}\ s_0\ \sim\ t_0\ \overset{!}{\leftarrow}\ t \tag{6.2}$$

which means that one first normalises the terms s, t to the unique normal forms s_0, t_0 by FOLDr and then compares them by the equality $\sim$. The reasons why this method is possible are

(i) $\to$ is SN,
(ii) $\to$ has the unique normal form property
(iii) $\to^*$ preserves $\sim$ by $\mathrm{CR}_\sim$, and
(iv) the bisimulation $\sim$ is decidable.

Hence (6.2) gives a decidable proof method for the equational theory generated by FOLDr, AxCy and AxBr.

6.5. **Establishing $\mathrm{CR}_\sim$.** Unfortunately, $\mathrm{CR}_\sim$ does not hold on unrestricted terms. For example, letting $e \triangleq (l, t.\, 2 :: t)$, we have the following situation, which violates $\mathrm{CR}_\sim$. A waved line in the diagram denotes $\sim$.

$$\begin{tikzcd} & 1 :: \mathsf{fold}(e, \mathsf{cy}(x.1 :: \mathsf{fold}(e,x))) \arrow[dl, "(4r)(5r)"'] \arrow[dr, squiggly, dash, "\text{(dinat)(sub)}"] & \\ 1 :: \mathsf{cy}(x.2 :: \mathsf{fold}(e, \mathsf{fold}(e,x))) \arrow[d, "/" marking] & \not\sim & \mathsf{cy}(x.1 :: \mathsf{fold}(e,x)) \arrow[d, "/" marking] \\ {} & & {} \end{tikzcd} \tag{6.3}$$

Note that the left-hand side rewrite applies (4r)(5r), which pushes fold into the constructor cy, and changes 1 to 2, because of the definition of e. A crucial point is that $\mathsf{cy}(x.1 :: \mathsf{fold}(e,x))$ involves a normal form $\mathsf{fold}(e,x)$, where *x is bound by a binder placed upward*. There are three possibilities that such a bound variable x appears:

(a) $\mathsf{cy}(\overrightarrow{y}.C[\ \mathsf{fold}(e,\ \overrightarrow{z}.\, x; \overrightarrow{z})\ ])$

(b) $(\overrightarrow{y}.C[\ \mathsf{fold}(e,\ \overrightarrow{z}.\, x; \overrightarrow{z})\ ]) \diamond t$

(c) $\mathsf{fold}(e, \overrightarrow{y}.C[\ \mathsf{fold}(e,\ \overrightarrow{z}.\, x; \overrightarrow{z}))\ ])$

where *a variable x is bound by one of binders $\overrightarrow{y}$*, and C is a context that does not bind x. Now $e = \overrightarrow{x_1}.e_1, \ldots, \overrightarrow{x_m}.e_m$. The cases (b) and (c) are no problem, i.e., they do not fall into a situation as the diagram (6.3). The only problematic case is (a). Hence, we exclude such terms from consideration.

Definition 6.6. Given a term t, if $\mathbf{nf}(t)$ has a subterm of the form

$$\mathsf{cy}(\overrightarrow{y}.C[\ \mathsf{fold}(e,\ \overrightarrow{z}.\, s; \overrightarrow{z}))\ ]),$$

where s involves a variable x bound by one of binders $\overrightarrow{y}$ of cy, then we call t a *bad term*. We define a set of non-bad terms as follows:

$\mathcal{T} \triangleq \{t \mid \Gamma \vdash t\ : \overrightarrow{c}$ with t is not bad, and every first m-arguments e of fold are closed $\}$.

The condition that the first m-arguments e of fold (cf. 4.2) are closed is to avoid similar problems.

Proposition 6.7. The set $\mathcal{T}$ is closed under the rewrite relation $\to_{\mathsf{FOLDr}}$ and one step application of an axiom in AxCy and AxBr.

Proof. If $t \in \mathcal{T}$ then $t \to_{\mathsf{FOLDr}} t' \in \mathcal{T}$ because FOLDr does not produce a bad term if t does not involve a bad term. Similarly, an application of an axiom in AxCy and AxBr does not produce a bad term if t does not involve a bad term. Note that substitution of terms for variables are always capture-avoiding. $\square$

Note that a bad term does not simply mean that fold does not appear under cy. A term where fold appears under cy is allowed, when s does not involve the bound variables $\overrightarrow{y}$ of cy. For example, the right-hand side of the rule (4r) in FOLDr is not bad.

We use the following theorem to show $\mathrm{CR}_\sim$.

Theorem 6.8. [AT12, Cor. 2.3] Let $\sim$ be an equivalence relation and $\to$ a binary relation on the same set. Suppose $\vdash\!\dashv$ is a symmetric relation such that $\vdash\!\dashv^* = \sim$. If

(i) $\to$ is well-founded,
(ii) $\leftarrow \circ \to \;\subseteq\; \to^* \circ \overset{=}{\vdash\!\dashv} \circ \leftarrow^*$ (i.e., $\to$ is *locally confluent in one step*), and
(iii) $\leftarrow \circ \vdash\!\dashv \;\subseteq\; \to^* \circ \overset{=}{\vdash\!\dashv} \circ \leftarrow^*$ (i.e., $\to$ is *locally* coherent *in one step*)

then $\to$ is Church-Rosser modulo $\sim$.

We apply the above theorem to our situation. We now take $\to$ to be $\to_{\mathsf{FOLDr}}$, $\sim$ to be the bisimulation restricted on $\mathcal{T}$, and $\vdash\!\dashv$ to be a relation on $\mathcal{T}$ defined by:

$s \vdash\!\dashv t$ iff $\Theta \vdash s = t : \overrightarrow{c}$ is derived from $\mathsf{AxCy} \cup \mathsf{AxBr}$ for some Θ and $\overrightarrow{c}$
by the cartesian second-order equational logic in Fig. 3
without using (Ref) and (Tra).

Namely, the symmetric relation $\vdash\!\dashv$ is the congruence closure of one-step application of an instance of an axiom of $\mathsf{AxCy} \cup \mathsf{AxBr}$ on $\mathcal{T}$. Thus $\vdash\!\dashv^* = \sim$. The condition (i) of Thm. 6.8 holds by Thm. 5.9. We check the conditions (ii) and (iii) of Thm. 6.8.

Proposition 6.9. The relations $\to_{\mathsf{FOLDr}}$ on $\mathcal{T}$ is locally confluent in one step.

Proof. Because $\to_{\mathsf{FOLDr}}$ is confluent. $\square$

Proposition 6.10. The relation $\to_{\mathsf{FOLDr}}$ on $\mathcal{T}$ is locally coherent in one step.

Proof. Let $\to$ denote $\to_{\mathsf{FOLDr}}$ on $\mathcal{T}$. We need to show that for an instance $s = t$ of each axiom such that s has a reduct s', s' and t commute modulo $\sim$. Thus we check all possible cases of the form $s' \leftarrow s \vdash\!\dashv t$. This is by induction on the proof of $s \vdash\!\dashv t$. Throughout the proof, we write $t\,\{\overrightarrow{s}\}$ for $t\{\overrightarrow{y} \mapsto \overrightarrow{s}\}$. Let $\overrightarrow{y} = y_1, \ldots, y_n$, $\overrightarrow{s} = s_1, \ldots, s_n$.

- (Ax1)(Ax2): We check each axiom in $\mathsf{AxCy} \cup \mathsf{AxBr}$.
  - (sub): Case $(\overrightarrow{y}.t) \diamond \langle \overrightarrow{s} \rangle = t\,\{\overrightarrow{s}\}$.
    - Case t is rewritten. We have

$$\begin{array}{ccc} (\overrightarrow{y}.t) \diamond \langle \overrightarrow{s} \rangle & \longmapsto\!\dashv & t\,\{\overrightarrow{s}\} \\ \downarrow & & \downarrow \\ (\overrightarrow{y}.t') \diamond \langle \overrightarrow{s} \rangle & \longmapsto\!\dashv & t'\,\{\overrightarrow{s}\} \end{array}$$

      The case s_i is rewritten is similar.
    - If the root of $(\overrightarrow{y}.t) \diamond \langle \overrightarrow{s} \rangle$ is rewritten, then it is finally rewritten to $t\,\{\overrightarrow{s}\}$.

- (sub) the converse: $t\,\{\overrightarrow{s}\} = (\overrightarrow{y}.t) \diamond \langle \overrightarrow{s} \rangle$.
  If $\overrightarrow{s}$ or t is rewritten, similarly to the above case.
  If $\overrightarrow{s}$ and t cannot be rewritten, but $t\,\{\overrightarrow{s}\}$ can be rewritten. Then there exist $1 \le i \le n$ and a context C such that $t = C[\mathsf{fold}(\overrightarrow{z}.y_i)] \quad (y_i \notin \overrightarrow{z})$. Then $t\,\{\overrightarrow{s}\} = C[\mathsf{fold}(\overrightarrow{z}.s_i)]$, s_i is one of the patterns in FOLDr, i.e., $\mathsf{fold}(\overrightarrow{z}.s_i) \to_{\mathsf{FOLDr}} r$, where $\overrightarrow{z}$ do not appear in s_i. We have

$$\begin{tikzcd}
C'[\mathsf{fold}(\overrightarrow{z}.s_i)] \arrow[r, |-|] \arrow[d] & (\overrightarrow{y}.C[\mathsf{fold}(\overrightarrow{z}.y_i)]) \diamond \langle \overrightarrow{s} \rangle \arrow[d, "+"] \\
C'[r] & C'[\mathsf{fold}(\overrightarrow{z}.s_i)] \arrow[l]
\end{tikzcd}$$

  where C' is obtained from C replacing every y_i in C with s_i for $1 \le i \le n$.

- (SP): $\langle (\overrightarrow{y}.y_1) \diamond t, \cdots, (\overrightarrow{y}.y_n) \diamond t \rangle = t$.
  - $*$ Case t is rewritten.

$$\begin{tikzcd}
\langle (\overrightarrow{y}.y_1) \diamond t, \cdots, (\overrightarrow{y}.y_n) \diamond t \rangle \arrow[r, |-|] \arrow[d] & t \arrow[dd] \\
\langle (\overrightarrow{y}.y_1) \diamond t', \cdots, (\overrightarrow{y}.y_n) \diamond t \rangle \arrow[d, "+"] & \\
\langle (\overrightarrow{y}.y_1) \diamond t', \cdots, (\overrightarrow{y}.y_n) \diamond t' \rangle \arrow[r, |-|] & t'
\end{tikzcd}$$

    The converse is similar.
  - $*$ Case $t = \langle \overrightarrow{u} \rangle$ and $(\overrightarrow{y}.y_i) \diamond t$ is rewritten. Then $\langle \overrightarrow{u} \rangle \leftarrow^{+} \langle \cdots, (\overrightarrow{y}.y_i) \diamond t, \cdots \rangle \mathrel{\vdash\!\dashv} \langle \overrightarrow{u} \rangle$.

- All the axioms of AxBr, (Bekič), (CI) and their converses: since the roots of the left and right-had sides of each axiom are not redexes, possible redexes appear only at the positions of metavariables (such as S, T) in the axiom, hence these are proved similarly to (SP).

- (dinat$_1$): $\mathsf{cy}(x.s\,\{z \mapsto t\}) = s\,\{z \mapsto \mathsf{cy}(z.t\,\{x \mapsto s\})\}$.
  - (i) Case s or t is rewritten. Similarly to the case (SP).
  - (ii) Case s and t cannot be rewritten, and $t\,\{x \mapsto s\}$ can be rewritten. This is only when $s = C[\mathsf{fold}(\overrightarrow{y}.z)]$ and t is one of the patterns in FOLDr. Then
    - $*$ $s\,\{t\} = C[\mathsf{fold}(\overrightarrow{y}.t)]$ and
    - $*$ $t\,\{s\} = t\{C[\mathsf{fold}(\overrightarrow{y}.x)]\}$.

    In this case, (dinat$_1$)'s rhs has a sub-term $\mathsf{cy}(z.t\{s\}) = \mathsf{cy}(z.s\,\{C[\mathsf{fold}(\overrightarrow{y}.z)]\})$, which is bad. Therefore, this case is excluded.

- (dinat$_n$): $\mathsf{cy}(\overrightarrow{x}.s\,\{\overrightarrow{\pi_i \diamond t}\}) = (\overrightarrow{z}.s) \diamond \mathsf{cy}(\overrightarrow{z}.t\,\{\overrightarrow{\pi_i \diamond s}\})$, where $\pi_i = \overrightarrow{y}.y_i$.
  - (i) Case s or t is rewritten. Similarly to the case (SP).
  - (ii) Case t and s cannot be rewritten. If $s = \langle s_1, \ldots, s_n \rangle$ and $t = \langle t_1, \ldots, t_n \rangle$, reducing $\pi_i \diamond s$ and $\pi_i \diamond t$, similarly to (ii). Otherwise, $\mathsf{cy}(\overrightarrow{x}.s\,\{\overrightarrow{\pi_i \diamond t}\})$ is not reducible, hence done.

- (dinat$_1$)(dinat$_n$): the converses are similar.

- (Fun) and (OSub): By induction hypothesis and the fact that rewriting and equational reasoning are closed under contexts and substitutions for variables. □

Theorem 6.11. FOLDr is $\mathrm{CR}_\sim$ on $\mathcal{T}$.

Proof. The relation $\to_{\mathsf{FOLDr}}$ on $\mathcal{T}$ is SN (Thm. 5.9), locally confluent in one step (Prop. 6.9), locally coherent in one step (Prop. 6.10). By Thm. 6.8, we have $\mathrm{CR}_\sim$. □

Remark 6.12. There are several other criteria to establish $\mathrm{CR}_\sim$, which requires termination (SN) of rewriting modulo the equivalence relation defined by $\to/\sim \;\triangleq\; (\sim\cdot\to\cdot\sim)$ [Hue80, JKR83] (see also [AT12, Thm. 2.2] for a unified theorem). However, in the case of FOLDr, the rewriting modulo bisimulation $\to/\sim$ is not SN because (dinat) can copy a redex inside cy. We write $\mathsf{fold}\,\langle\,\rangle$ for $\mathsf{fold}(e,\langle\,\rangle)$. For example, since $\mathsf{fold}\,\langle\,\rangle$ is a redex by (2r) (N.B. $\mathsf{cy}(y.\mathsf{fold}\,\langle\,\rangle)$ is not bad), we have an infinite derivation

$$\begin{aligned}
\mathsf{cy}(y.\mathsf{fold}\,\langle\,\rangle) &\sim \underline{(y.\mathsf{fold}\,\langle\,\rangle)} \diamond \mathsf{cy}(y.\mathsf{fold}\,\langle\,\rangle)\\
&\to (y.\langle\,\rangle) \diamond \underline{\mathsf{cy}(y.\mathsf{fold}\,\langle\,\rangle)}\\
&\sim (y.\langle\,\rangle) \diamond \underline{(y.\mathsf{fold}\,\langle\,\rangle)} \diamond \mathsf{cy}(y.\mathsf{fold}\,\langle\,\rangle)\\
&\to (y.\langle\,\rangle) \diamond (y.\langle\,\rangle) \diamond \underline{\mathsf{cy}(y.\mathsf{fold}\,\langle\,\rangle)} \;\sim \cdots
\end{aligned}$$

where each underlined term is transformed.

As discussed in §6.4, we have a remarkable result.

Corollary 6.13. The equational theory generated by FOLDr, AxCy and AxBr on terms of $\mathcal{T}$ is decidable.

Since FOLDr is SN and $\mathrm{CR}_\sim$, we have the following fundamental property.

Proposition 6.14. If $\overrightarrow{z:b},\Gamma \vdash s \;:\overrightarrow{c}$ and $\overrightarrow{z:b},\Gamma \vdash t \;:\overrightarrow{c}$ are not bad, then

$$s \sim t \quad\Rightarrow\quad \mathsf{fold}(\overrightarrow{e},\, x_1,\ldots,x_n.s;\; \overrightarrow{x}) \overset{!}{\to} \circ \sim \circ \overset{!}{\leftarrow} \mathsf{fold}(\overrightarrow{e},\, x_1,\ldots,x_n.t;\; \overrightarrow{x})$$

where $\Gamma = x_1 : b_1,\ldots,x_n : b_n$.

Proof. If $s \sim t$, then $\mathsf{fold}(\overrightarrow{e},\, x_1,\ldots,x_n.s;\; \overrightarrow{x}) \sim \mathsf{fold}(\overrightarrow{e},\, x_1,\ldots,x_n.t;\; \overrightarrow{x})$, hence their normal forms are bisimilar by $\mathrm{CR}_\sim$. □

Remark 6.15. We introduced the notion of bad terms to establish the property $\mathrm{CR}_\sim$. Another explanation of bad terms is that a bad term generates a non-rational tree. For example, consider a function incrementing each element of a `CNat`-list defined by

$$\mathtt{mapinc}(t) = \mathsf{fold}([\,],\; \ell.s.\ell+1 :: s,\; t)$$

This function itself is no problem. For example, applying `mapinc` to the cyclic list of 1, we have the cyclic list of 2.

```
mapinc(cy(x. 1:: x)) → cy(x. mapinc(x. 1:: x; x))
→ cy(x. 2 :: mapinc(x; x)) → cy(x. 2:: x)
```

But consider a *bad term* $\mathsf{cy}(z.1 :: \mathtt{mapinc}(z))$. Observe that it is a *normal form* with respect to FOLDr, because $\mathtt{mapinc}(z)$ cannot be rewritten. But using the cartesian second-order equational logic with AxCy (especially (fix)) and FOLD, we can reason as follows:

$$
\begin{aligned}
\mathsf{cy}(z.1 :: \mathtt{mapinc}(z)) &= 1 :: \mathtt{mapinc}(\mathsf{cy}(z.1 :: \mathtt{mapinc}(z))) \\
&= 1 :: \mathtt{mapinc}(1 :: \mathtt{mapinc}(\mathsf{cy}(z.1 :: \mathtt{mapinc}(z)))) \\
&= 1 :: (2 :: \mathtt{mapinc}(\mathtt{mapinc}(\mathsf{cy}(z.1 :: \mathtt{mapinc}(z))))) \\
&= 1 :: 2 :: 3 :: \cdots :: n :: \mathtt{mapinc}^n(\mathsf{cy}(z.1 :: \mathtt{mapinc}(z)))
\end{aligned}
\tag{6.4}
$$

which essentially means that it corresponds to a non-rational tree. Any cyclic term *without fold* is interpreted as a rational tree in **CPO** because a free iteration theory of binary branching trees by AxBr characterises rational trees modulo bisimulation [Ési00, BÉ93] (see also [Ham15, Thm. 3.2]). Then the continuous function **fold** in **CPO** is a function between rational trees modulo bisimulation, but the example (6.4) shows a non-rational tree. In this sense, we excluded the bad term from $\mathcal{T}$.

Note that a bad term $\mathsf{cy}(z.1 :: \mathtt{mapinc}(z))$ can be represented in Haskell as a recursive definition of list

```
z = 1:map (1+) z
```

which is a well-known lazy programming technique. It looks like this is defining a cyclic data, but actually defining an acyclic infinite data described in [FS96, §2.2.1], and analysed in our previous work [GHUV06, §2.1]. We could capture this phenomenon from a different viewpoint, namely from the viewpoint of rewriting. From the view of functional programming, bad terms generate acyclic infinite data, and from the view of rewriting, bad terms are terms preventing $\mathrm{CR}_\sim$.

7. Computing by Fold on Cyclic Datatypes

In this section, we demonstrate fold computation on cyclic data by several examples. In this section, we write $\to$ to mean $\to_{\mathsf{FOLDr}\cup\mathsf{SIMP}}$.

Example 7.1. We consider the emptiness check of cyclic sharing trees. It means to check whether a given closed term is bisimilar to $[]$. This is not just to check whether a given term is syntactically $[]$. For example, is `cy(x.x) + cy(x.[] + x)` empty? This requires certain computation, which we will define. First, we define a (cyclic) boolean datatype having the operation $\wedge$, satisfying the axioms AxBr.

```
ctype Bool where                isEmpty : CTree → Bool
  true  : Bool                  spec isEmpty([])     = true
  false : Bool                       isEmpty(f(t))  = false
  ∧   : Bool,Bool → Bool             isEmpty(s + t) = isEmpty(s) ∧ isEmpty(t)
with axioms AxCy,AxBr(true,∧)

fun isEmpty(t) = fold (true, x.false, x.y.x ∧ y) t
```

We define `isEmpty` by fold. Examples of computation are as follows, where the cycle cleaning laws in SIMP are crucial.

```
  isEmpty(cy(x.x) + cy(x.[]+x))
→+ isEmpty([]+cy(x.x))
→ isEmpty([]+[])
→+ true ∧ true → true
```

```
isEmpty( cy(x^Ctree.cy(w.x)) )
→ cy(x^Bool. isEmpty(x.cy(w.x); x))
→ cy(x^Bool. cy( w.isEmpty(x.w.x; x,w) ))
→ cy(x^Bool. cy( w.x ))
→ cy(x^Bool.x) → true
```

```
isEmpty(cy(x. a(cy(y.y+y)) + cy(w.x)))
→+ isEmpty(cy(x. a([]) + x))
→ isEmpty(a([])) → false
```

The example `isEmpty(cy(x.x) + cy(x.[]+x))` is essentially checking null-ability of a grammar [Brz64], which was pointed as a difficult problem for computing with cyclic data [OC12]. Our rewrite system FOLDr with SIMP successfully computes it in a principled manner (i.e., without any special treatment).

Example 7.2. 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 a `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 [GHUV06]. A naive unfolding by using the fixed point law $\mathsf{cy}(x.t) = t\ \{x \mapsto \mathsf{cy}(x.t)\}$ violates strong normalisation because it copies the original term. It actually *increases* complexity.

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

We define `ctail` by fold. Rather than the fixed point law, another important principle of cyclic structures of *Bekič law* plays a crucial role here.

```
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 7.3. We define the datatype of cyclic strings consisting of `a`, `b`, the empty string ε, and the choice operator "`|`".

```
ctype CString where
  a : CString → CString
  b : CString → CString
  ε : CString
  |  : CString,CString → CString
with axioms AxCy,AxBr(ε,|)
```

We consider the function `aa?` that checks whether a given closed term contains two consecutive `a`'s, such as $\mathtt{a(a(\cdot))}$. For example,

$$
\begin{array}{lcl}
\mathtt{aa?(\ b(a(a(b(}\varepsilon\mathtt{))))\ )} & \xrightarrow{!} & \mathtt{true} \\
\mathtt{aa?(\ b(b(}\varepsilon\mathtt{))} & \xrightarrow{!} & \mathtt{false} \\
\mathtt{aa?(\ b(}\varepsilon\mathtt{)|a(a(}\varepsilon\mathtt{))\ )} & \xrightarrow{!} & \mathtt{true}
\end{array}
$$

Even in case of a string containing cycle, the function must return a correct result. For example, we expect the following results.

$$
\begin{array}{lcl}
\mathtt{aa?(\ cy(x.a(x))\ )} & \xrightarrow{!} & \mathtt{true} \\
\mathtt{aa?(\ b(cy(x.a(b(a(x)))))\ )} & \xrightarrow{!} & \mathtt{true}
\end{array}
$$

We now specify `aa?` by using the function `head-a?` which checks whether the head is `a`.

$$
\begin{array}{ll}
\mathtt{head\text{-}a?\ :\ CString} \to \mathtt{ABool} & \mathtt{aa?\ :\ CString} \to \mathtt{ABool} \\
\mathtt{spec\ head\text{-}a?(a(t))\ =\ true} & \mathtt{spec\ aa?(a(t))\ =\ head\text{-}a?(t)} \\
\qquad \mathtt{head\text{-}a?(b(t))\ =\ false} & \qquad \mathtt{aa?(b(t))\ =\ aa?(t)} \\
 & \qquad \mathtt{aa?(}\ \varepsilon\ \mathtt{)\ =\ false} \\
 & \qquad \mathtt{aa?(s\ |\ t)\ =\ aa?(s)} \vee \mathtt{aa?(t)}
\end{array}
$$

The results of `aa?` must be boolean, but it should not be the type `Bool` defined in Example 7.1. Now we need the type `ABool` of "additive" boolean, meaning that it has the conjunction "$\vee$" with the unit `false`, because the specification of `aa?` requires that the unit ε (resp. the multiplication "`|`") of `CString` is mapped to the unit `false` (resp. the multiplication "$\vee$") of the target type.

$$
\begin{array}{l}
\mathtt{ctype\ ABool\ where} \\
\quad \mathtt{true}\ \ \ \mathtt{:\ ABool} \\
\quad \mathtt{false\ :\ ABool} \\
\quad \vee\ \ \ \ \ \ \mathtt{:\ ABool,ABool} \to \mathtt{ABool} \\
\mathtt{with\ axioms}\ \mathsf{AxCy},\mathsf{AxBr}\mathtt{(false,}\vee\mathtt{)}
\end{array}
$$

Then we can define the functions `head-a?` and `aa?` by fold.

$$
\begin{array}{l}
\mathtt{fun}\ \ \mathtt{head\text{-}a?(t)\ =\ fold\ (x.true,x.false,false,x.y.x}\vee\mathtt{y)\ t} \\
\mathtt{fun}\ \ \mathtt{aa?(t)\ =}\ \pi_1 \diamond \mathtt{fold\ (v,w.\langle head\text{-}a?(w),a(w)\rangle,\ v,w.\langle v,b(w)\rangle} \\
\qquad\qquad\qquad\qquad\quad \mathtt{\langle false,}\varepsilon\mathtt{\rangle,\ v_1.w_1.v_2.w_2.\langle v_1}\vee\mathtt{v_2,w_1|w_2\rangle)\ t}
\end{array}
$$

We demonstrate how our system correctly computes

$$\mathtt{aa?(cy(x.a(x)))} \quad \xrightarrow{!} \quad \mathtt{true}.$$

One may think that it needs expansion of the inner term of `cy`. We show that the rewrite system $\mathsf{FOLDr} \cup \mathsf{SIMP}$ does it *without using the fixed point law*, rather, by using *Bekič law*. We write `folde` for `fold (v.w.<head-a?(w),a(w)>, v.w.<v,b(w)>,...)`.

$$
\begin{array}{l}
\mathtt{aa?(cy(x.a(x)))\ =}\ \pi_1 \diamond \mathtt{folde\ cy(x.a(x))} \\
\ \to \pi_1 \diamond \mathtt{cy(v.w.(folde\ a(v);\ v,w))} \\
\ \to^+ \pi_1 \diamond \mathtt{\langle cy(v.head\text{-}a?(cy(w.a(w)))),\ cy(w.a(w))\rangle} \\
\ \to^+ \mathtt{head\text{-}a?(cy(w.a(w))} \to \mathtt{cy(w.head\text{-}a?(w.a(w);w)\ )} \to^+ \mathtt{cy(w.true)} \to \mathtt{true}
\end{array}
$$

Again, the highlighted step uses Bekič law. Moreover, we expect

$$\texttt{aa?(cy(x.b(x)))} \quad \stackrel{!}{\to} \quad \texttt{false}$$

without falling into non-termination. This is obtained without any special treatment.

$$\begin{aligned}
&\texttt{aa?(cy(x.b(x))) = } \pi_1 \diamond \texttt{ folde cy(x.b(x))} \\
&\ \to \pi_1 \diamond \texttt{ cy(v.w. (folde b(v); v,w))} \\
&\to^{+} \pi_1 \diamond \texttt{ <cy(v.v),cy(w.b(w))>} \to^{+} \texttt{cy(v.v)} \to \texttt{false}
\end{aligned}$$

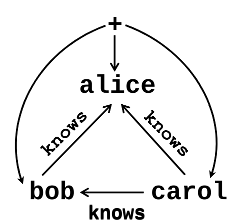
A crucial point is that since we chose the target type as `ABool`, the rule (14r) in FOLDr∪SIMP rewrites `cy(v.v)` to `false`, which is the unit of `ABool`. If we chose the target type as `Bool` in Example 7.1, this would become an incorrect result `true`. But the specification forced us to choose `ABool`.

Example 7.4. This example shows that our cyclic datatype has ability to express graphs. The graph shown below right represents friend relationship, which describes Bob knows Alice, and Carol knows Alice and Bob. To make it a rooted graph, it has the uppermost node "+" which points to the nodes of three persons. This is represented as a term

$$\texttt{(a.b.c.a+b+c)} \diamond \texttt{ cy(a.b.c.<name("alice"), name("bob")+knows(a),}$$
$$\texttt{name("carol")+knows(a)+knows(b)>)}$$

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)
```

$$\begin{aligned}
&\texttt{fun collect t = } \pi_1 \diamond \texttt{ folde (x,y.<x,knows(y)>, x,y.<nm(y),name(y)>,} \\
&\qquad\qquad\qquad\qquad \texttt{<[],[]>, } \mathtt{v_1.w_1.v_2.w_2.<v_1{+}v_2,w_1{+}w_2>}\texttt{) t}
\end{aligned}$$

Then we collect certainly all names by our system 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 ◇ (a.a'.b.b'.c.c'.<a+b+c,a'+b'+c'>) ◇ (cy(a.a'.b.b'.c.c'.
    <folde(a.b.c.name("alice")), folde(a.b.c.name("bob")+knows(a)),
     folde(a.b.c.name("carol")+knows(a)+knows(b)>)))
→+ π₁ ◇ (a.a'.b.b'.c.c'.<a+b+c,a'+b'+c'>) ◇ (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")
```

8. Related work

8.1. **Cyclic structures represented by systems of equations.** Other than the term representation used in the present paper, various representations of cyclic structures or graphs are known. One of the frequently used representations is by a *system of equations*, also known as an equational term graph [AK96]. It is essentially the same as the representation of cyclic structures using **letrec**-expressions used in [Has97]. Semantically, the least solution of a system of equations is regarded as a (unfolded) graph. Syntactically, a system of (flat) equations can be seen as adjacency lists of a graph, e.g. an equation $x = f(y_1, \ldots, y_n)$ in a system can be regarded as an adjacency list of vertex x pointed to by other vertices $y_1, \ldots, y_n$ (cf. the discussions in [Ham10, Sect. 8]). These representations and our term representation are equivalent. For example, a system of equations

$$\left\{\begin{array}{l} x_1 = f(x_1, x_2), \\ x_2 = g(x_1) \end{array}\right\}$$

is equivalently represented as a term

$$\mathsf{cy}(x_1, x_2.\langle f(x_1, x_2), g(x_1)\rangle).$$

Nishimura and Ohori [Nis97, NO99] developed a general mechanism for programming with cyclic structures in a purely functional style using the representation of system of equations. They developed the "reduce" operation on cyclic structures based on a mechanism for data-parallelism on recursive data, which is fold in our sense, although bisimilar cyclic structures are not identified in their work.

8.2. **Foundational graph rewriting calculi.** There has been various work to deal with graph computation and cyclic data structures in functional programming and foundational calculi including [FS96, GHUV06, Ham09, Ham10, Ham12, OC12, HHI+10, MA15, AK96, AB97]. Foundational graph rewriting calculi, such as equational term graph rewriting systems [AK96], are general frameworks of graph computation. The fold on a cyclic datatype in this paper is more restricted than general graph rewriting. However, our emphasis is clarification of the categorical and algebraic structures of cyclic datatypes and the computation by fold on them by regarding fold as a structure preserving map, rather than unrestricted rewriting. This was a key to obtain SN and $\mathrm{CR}_\sim$ of FOLDr. We also hope that it will be useful for further optimisation such as the fold fusion based on semantics as done in [HMA, §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 [AK96], due to [BÉ93]. This observation has been used to give an effective operational semantics of graph transformation in [MA15].

9. Conclusion

In this paper, we have developped foundations of cyclic datatypes and computation:

[I] Syntax and type system supporting algebraic datatypes with cycle and sharing constructs (§2)

[II] Complete equational axioms for bisimulation of cyclic data (Fig. 4)

[III] Algebraic theory FOLD of fold on cyclic datatypes and its strongly normalising rewrite system FOLDr (Fig. 7, Fig. 6) (§4, §5), which is Church-Rosser modulo bisimulation (§6)

[IV] The framework that supports [I]-[III] based on cartesian second-order algebraic theory (§2.1) and iteration category (§3). The numbered items [I]-[III] in Fig. 1 in Introduction are instances of these results.

We have not assumed 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 assistances requiring terminating functions, such as Coq or Agda. We have shown several concrete examples of cyclic data computation in §7. In this paper, we have focused on the underlying theory of cyclic datatypes. Formal development of the programming language that realises the program codes described in this paper is left for a future work.

9.1. **Acknowledgments.** I am grateful to Kazutaka Matsuda and Kazuyuki Asada for various discussions about calculi and programming languages about graphs and cyclic structures, and Frédéric Blanqui for clarifying some details of the General Schema. I also thank the reviewers for their positive and constructive comments.

References

[AB97] Z. M. Ariola and S. Blom. Cyclic lambda calculi. In *Theoretical Aspects of Computer Software, LNCS 1281*, pages 77–106, 1997.

[Acz78] P. Aczel. A general Church-Rosser theorem. Technical report, University of Manchester, 1978.

[AJ94] S. Abramsky and A. Jung. Domain theory. In D. Gabbay and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 3, pages 1–168. Oxford University Press, 1994.

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

[AT12] T. Aoto and Y. Toyama. A reduction-preserving completion for proving confluence of non-terminating term rewriting systems. *Logical Methods in Computer Science*, 8(1), 2012.

[BÉ93] S. L. Bloom and Z. Ésik. *Iteration Theories – The Equational Logic of Iterative Processes.* EATCS Monographs on Theoretical Computer Science. Springer, 1993.

[BFS00] 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.

[BJO02] F. Blanqui, J.-P. Jouannaud, and M. Okada. Inductive data type systems. *Theoretical Computer Science*, 272:41–68, 2002.

[Bla00] F. Blanqui. Termination and confluence of higher-order rewrite systems. In *Rewriting Techniques and Application (RTA 2000)*, LNCS 1833, pages 47–61. Springer, 2000.

[Bla16] F. Blanqui. Termination of rewrite relations on λ-terms based on Girard's notion of reducibility. *Theor. Comput. Sci.*, 611:50–86, 2016.

[BN98] F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.

[Brz64] J. A. Brzozowski. Derivatives of regular expressions. *J. ACM*, 11, 1964.

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

[Chl08] A. Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In *ICFP'08: Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, 2008.

[Coq92] T. Coquand. Pattern matching with dependent types. In *Proc. of the 3rd Work. on Types for Proofs and Programs*, 1992.

[Cou83] B. Courcelle. Fundamental properties of infinite trees. *Theoretical Computer Science*, 25(2):95–169, 1983.

[DFH95] 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.

[DPP04] A. Dovier, C. Piazza, and A. Policriti. An efficient algorithm for computing bisimulation equivalence. *Theoretical Computer Science*, 311:221–256, 2004. Issues 1-3.

[Ési99] Z. Ésik. Axiomatizing iteration categories. *Acta Cybernetica*, 14:65–82, 1999.

[Ési00] Z. Ésik. Axiomatizing the least fixed point operation and binary supremum. In *Proc. of Computer Science Logic 2000*, LNCS 1862, pages 302–316, 2000.

[FC13] 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.

[FH10] M. Fiore and C.-K. Hur. Second-order equational logic. In *Proc. of CSL'10*, LNCS 6247, pages 320–335, 2010.

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

[FM10] M. Fiore and O. Mahmoud. Second-order algebraic theories. In *Proc. of MFCS'10*, LNCS 6281, pages 368–380, 2010.

[FPT99] M. Fiore, G. Plotkin, and D. Turi. Abstract syntax and variable binding. In *Proc. of LICS'99*, pages 193–202, 1999.

[FS96] L. Fegaras and T. Sheard. Revisiting catamorphisms over datatypes with embedded functions (or, programs from outer space). In *POPL'96*, pages 284–294, 1996.

[FS14] M. Fiore and S. Staton. Substitution, jumps, and algebraic effects. In *Proc. of LICS'14*, 2014.

[GHUV06] N. Ghani, M. Hamana, T. Uustalu, and V. Vene. Representing cyclic structures as nested datatypes. In *Proceedings of Trends in Functional Programming*, pages 173–188, 2006.

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

[Ham05] M. Hamana. Universal algebra for termination of higher-order rewriting. In *Proc. of RTA'05*, LNCS 3467, pages 135–149, 2005.

[Ham07] M. Hamana. Higher-order semantic labelling for inductive datatype systems. In *Proc. of PPDP'07*, pages 97–108. ACM Press, 2007.

[Ham09] M. Hamana. Initial algebra semantics for cyclic sharing structures. In *Proc. of TLCA'09*, LNCS 5608, pages 127–141, 2009.

[Ham10] M. Hamana. Initial algebra semantics for cyclic sharing tree structures. *Logical Methods in Computer Science*, 6(3), 2010.

[Ham12] M. Hamana. Correct looping arrows from cyclic terms: Traced categorical interpretation in Haskell. In *Proc. of FLOPS'12*, LNCS 7294, pages 136–150, 2012.

[Ham15] 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.

[Ham16] M. Hamana. Strongly normalising cyclic data computation by iteration categories of second-order algebraic theories. In *Proc. of The 1st International Conference on Formal Structures for Computation and Deduction (FSCD'16)*, volume 52 of *the Leibniz International Proceedings in Informatics (LIPIcs)*, pages 21:1–21:18, 2016.

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

[HHI+10] 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.

[HMA] M. Hamana, K. Matsuda, and K. Asada. The algebra of recursive graph transformation language UnCAL: Complete axiomatisation and iteration categorical semantics. to appear in Mathematical Structures in Computer Science, Cambridge University Press.

[Hue80] G. Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *Journal of ACM*, 27(4):797–821, 1980.

[JKR83] J. Jouannaud, H. Kirchner, and J. Remy. Church-rosser properties of weakly terminating term rewriting systems. In *Proceedings of the 8th International Joint Conference on Artificial Intelligence. Karlsruhe, FRG, August 1983*, pages 909–915, 1983.

[JSV96] A. Joyal, R. Street, and D. Verity. Traced monoidal categories. *Mathematical Proceedings of the Cambridge Philosophical Society*, 119(3):447–468, 1996.

[Klo80] J.W. Klop. *Combinatory Reduction Systems*. PhD thesis, CWI, Amsterdam, 1980. volume 127 of Mathematical Centre Tracts.

[MA15] 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.

[Mee92] L. G. L. T. Meertens. Paramorphisms. *Formal Asp. Comput.*, 4(5):413–424, 1992.

[MFP91] 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.

[Mil84] R. Milner. A complete inference system for a class of regular behaviours. *J. Comput. Syst. Sci.*, 28(3):439–466, 1984.

[Nis97] S. Nishimura. A strict functional language with cyclic recursive data. *Formal Asp. Comput.*, 9(1):78–97, 1997.

[NO99] S. Nishimura and A. Ohori. Parallel functional programming on recursively defined data via data-parallel recursion. *J. Funct. Program.*, 9(4):427–462, 1999.

[OC12] B. C.d.S. Oliveira and W. R. Cook. Functional programming with structured graphs. In *Proc. of ICFP'12*, pages 77–88, 2012.

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

[SP00] A. K. Simpson and G. D. Plotkin. Complete axioms for categorical fixed-point operators. In *Proc. of LICS'00*, pages 30–41, 2000.

[Sta13] S. Staton. An algebraic presentation of predicate logic. In *Proc. of FOSSACS 201*, pages 401–417, 2013.

[Sta15] S. Staton. Algebraic effects, linearity, and quantum programming languages. In *Proc. of POPL'15*, pages 395–406, 2015.

[Ter03] Terese. *Term Rewriting Systems*. Number 55 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2003.

[Win93] G. Winskel. *The Formal Semantics of Programming Languages*. The MIT Press, 1993.