Functional Logic Programming with Generalized Circular Coinduction*

Ronald de Haan
Technische Universität Dresden

Abstract

We propose a method to adapt functional logic programming to deal with reasoning on coinductively interpreted programs as well as on inductively interpreted programs. In order to do so, we consider a class of objects interesting for this coinductive interpretation, namely regular terms. We show how the usual data structures can be adapted to capture these objects. We adapt the operational semantics of Curry to interpret programs coinductively. We illustrate this method with several examples that show the working of our method and several cases in which it could be useful. Finally, we suggest how the declarative semantics can be adapted suitably.

1 Introduction

Within the paradigm of declarative programming, several styles of programming have been developed, among which logic and functional programming. One of the benefits of logic programming is its use of free variables and its inherent capabilities to model search problems. The benefits of functional programming include its efficient execution. Attempts to combine the best of both worlds have resulted in functional logic programming. An example of this combination is the programming language Curry [1, 2].

Another development within the paradigm of declarative programming is to develop mechanisms to reason about coinductively defined semantics, besides the usual inductively defined semantics. This is useful, for instance, to reason about infinite data, and processes and their states. In the case of logic programming, this resulted in co-logic programming [3]. Another approach to reason about coinductive semantics concerns proof mechanisms based on circularity [4], which are used in the coinductive prove engine CIRC [5], for instance. However, in the case of functional logic programming, no such adaptation to a coinductive interpretation has been developed.

In this paper, we will generalize the idea of circular coinduction to be used in the domain of functional logic programming. Similarly to detecting circularity in equality proofs (as is the case in circular coinduction), we will detect circularity in a reduction derivation. By means of assuming a certain resulting value for terms that appear repeatedly in such circular reduction sequences, we can resolve these circularities. Circular coinduction is a particular (Boolean) case of this general mechanism. Hence the name generalized circular coinduction for this method.

In Section 2, we will define some preliminary notions used in our approach. We define a data structure to capture a particular (finitely representable) class of infinite terms, and we show how

*I would like to thank Prof. Dr. Horst Reichel for his useful input and comments.
to perform several basic operations on these data structures. Then, in Section 3 we describe how we propose to modify the computational strategy from [1] with generalized circular coinduction. In Section 4 we will show the benefits of this approach by means of several examples. We suggest a possible declarative semantics for the mechanism in Section 5 before concluding and suggesting directions of further research in Section 6.

2 Preliminaries

2.1 Cyclic terms

Certain classes of infinite terms can be described finitely. One such class is the class of regular infinite terms. A term is regular if it has only finitely many different subterms. It is easy to see that all finite terms are regular. In this paper we will consider (finite and infinite) finitely-branching regular terms, such as the terms in Example 1. Example 2 shows a term that is not finitely-branching.

Example 1. An example of a non-regular infinite term is the infinite tree $\Gamma_1$, where $\Gamma_i$ for $i \in \mathbb{N}$ is defined as follows, for a (fixed) binary constructor symbol $\ast$:

$$\Gamma_i = \ast(i, \Gamma_{i+1})$$

The set of subterms $\text{sub}(\Gamma_1) = \mathbb{N} \cup \{\Gamma_j \mid j \in \mathbb{N}\}$ of $\Gamma_1$ is infinite. An example of a regular infinite term is the infinite tree $\Delta$, where $\Delta$ is defined as follows, for a (fixed) binary constructor symbol $\ast$:

$$\Delta = \ast(1, \Delta)$$

The set of subterms $\text{sub}(\Delta) = \{1, \Delta\}$ of $\Delta$ is finite.

Example 2. Another example of a regular infinite term is the infinite tree $\Lambda$, defined as follows, for a (fixed) constructor symbol $\circ$ of infinite arity.

$$\Lambda = \circ(\Lambda, \Lambda, \ldots)$$

In other words, each of the (infinitely many) direct subterms of the root of $\Lambda$ is $\Lambda$. Note that the set of subterms $\text{sub}(\Lambda) = \{\Lambda\}$ of $\Lambda$ is finite. In this paper, however, we restrict ourselves to finitely branching terms.

In the following we develop the technical machinery to represent such (finitely-branching) regular terms. We will use two different kinds of variables: frozen variables and regular variables. We fix a set $Y = \{y, y', y'', \ldots\} \cup \{y_1, y_2, \ldots\}$ of frozen variables and a set $X = \{x, x', x'', \ldots\} \cup \{x_1, x_2, \ldots\}$ of regular variables. We let $\Sigma = C \cup F$ denote a signature consisting of constructor symbols $C$ and function symbols $F$. We let $\text{Var} : \Sigma(X \cup Y) \rightarrow P(X)$ denote the function that returns the set of all variables occurring in a given term. Also, we let $\text{FVar} : \Sigma(X \cup Y) \rightarrow P(Y)$ denote the function that returns the set of all frozen variables occurring in a given term. We denote the extension of $\text{Var}$ (resp. $\text{FVar}$) to sets of terms also with $\text{Var}$ (resp. $\text{FVar}$).

Consider a mapping $\rho$ from a set $S$ to the same set $S$, that maps all but a finite set of values $\{s_1, \ldots, s_n\} \subseteq S$ to themselves. In the following, we will interchangeably use the following three views of such mappings. Either we view $\rho$ as a finite set of pairs $\{s_1 \mapsto t_1, \ldots, s_n \mapsto t_n\}$ that correspond to the mappings of elements that are not mapped to themselves, or we view $\rho$ as a total function with domain $\{s_1, \ldots, s_n\}$, or we view $\rho$ as a total function with domain $S$, that maps all values not in $\{s_1, \ldots, s_n\}$ to themselves. Also, for any such mapping $\rho$ and any $S \subseteq \text{dom}(\rho)$, we let $\rho|_S = \{s \mapsto t \in \rho \mid s \in S\}$.

We represent finitely branching, regular terms with the following structures.
**Definition 3** (Cyclic terms). A cyclic term over a signature $\Sigma$ and a set of variables $\mathcal{X}$ is a term $t \in \Sigma(\mathcal{X} \cup \mathcal{Y})$, together with a complete mapping $\theta : F Var(t) \rightarrow \Sigma(\mathcal{X} \cup F Var(t)) \setminus F Var(t)$. We say that a cyclic term is in base form if $Var(rng(\theta)) \subseteq Var(t)$.

A cyclic term consisting of $t$ and $\theta$ is denoted with $(t, \theta)$. We denote the set of all cyclic terms over the signature $\Sigma$, the set of variables $\mathcal{X}$ and the set of frozen variables $\mathcal{Y}$ with $CT(\Sigma, \mathcal{X}, \mathcal{Y})$, or simply $CT$ if $\Sigma$, $\mathcal{X}$ and $\mathcal{Y}$ are clear from the context.

Example 4 illustrates how finitely-branching regular terms can be represented by cyclic terms.

**Example 4.** The term $\Delta$ from Example 1 with * being the list constructor operator $: -$ in other words, the infinite list containing only 1’s – can be represented as the cyclic term $(y, \{y \mapsto 1 : y\})$.

Certain cyclic terms that are syntactically different can intuitively represent the same terms. Below, we define a notion of equivalence of cyclic terms (based on bisimulation on labelled graphs extracted from cyclic terms) to capture this intuition. In Example 8 we give an example of two syntactically different, but equivalent cyclic terms.

**Definition 5** (Cyclic term graph). For a given cyclic term $(t, \theta)$ we define the cyclic term graph $(V, E, V\mathcal{L}, EL)$ as a graph $(V, E)$ together with a vertex labelling $V\mathcal{L} : V \rightarrow 2^\mathcal{Y}$ and an edge labelling $EL : E \rightarrow \mathbb{N}$ as follows:

- $V := (sub(t) \cup sub(rng(\theta))) \setminus \mathcal{Y}$;
- $y \in V\mathcal{L}(v)$ if $y \in \mathcal{Y}, v \in V$ and $y \mapsto v \in \theta$; and
- $(v, v') \in E$ and $EL(v, v') = n$ if
  - either $v'$ is the $n$-th subterm of $v$,
  - or there exists a $y \in \mathcal{Y}$ such that $y$ is the $n$-th subterm of $v$ and $v'$ is the unique vertex in $V$ such that $y \in V\mathcal{L}(v')$.

Here $sub$ denotes the function that returns the set of subterms occurring in a given term. Note that for every $y \in F Var(t)$ there exists a unique node $v \in V$ such that $y \in V\mathcal{L}(v)$, namely the unique term $v \in \Sigma(\mathcal{X} \cup \mathcal{Y})$ for which holds $y \mapsto v \in \theta$.

**Definition 6** (Cyclic term graph bisimulation). For two given cyclic term graphs $(V, E, V\mathcal{L}, EL)$ for the cyclic term $(t, \theta)$, and $(V', E', V\mathcal{L}', EL')$ for the cyclic term $(t', \theta')$, we say that two nodes $w \in V$ and $w' \in V'$ bisimulate if:

- there exists a relation $Z \subseteq V \times V'$ that satisfies the following conditions:
  - if $(v, v') \in Z$, then the terms $v$ and $v'$ must have the same outermost symbol from $\Sigma$, or be the same variable from $\mathcal{X}$;
  - if $(v_1, v'_1) \in Z$, and also $(v_1, v_2) \in E$ and $EL(v_1, v_2) = n$, then there must exist a $v'_2 \in V'$ such that $(v_2, v'_2) \in Z$, $(v_1, v'_2) \in E'$ and $EL'(v'_1, v'_2) = n$; and
  - if $(v_1, v'_1) \in Z$, and also $(v'_1, v'_2) \in E'$ and $EL'(v'_1, v'_2) = n$, then there must exist a $v_2 \in V$ such that $(v_2, v'_2) \in Z$, $(v_1, v'_2) \in E$ and $EL(v_1, v_2) = n$;
- $(w, w') \in Z$.

Note that this notion of bisimulation does not depend in any way on the vertex labelings $V$ and $V'$.

For any $y \in F Var(t)$ and $y' \in F Var(t')$, we say that $y$ and $y'$ bisimulate in the cyclic term graphs if we have that $u$ and $u'$ bisimulate in the cyclic term graphs, where $u \in V$ is the unique node such that $y \in V\mathcal{L}(u)$ and $u' \in V'$ is the unique node such that $y' \in V\mathcal{L}'(u')$. 

3
Definition 7 (Equivalence). Two cyclic terms \((t, \theta)\) and \((t', \theta')\) are equivalent, denoted with \((t, \theta) \equiv (t', \theta')\), iff \(t\) and \(t'\) bisimulate in the cyclic term graphs for \((t, \theta)\) and \((t', \theta)\).

Example 8. Consider the two cyclic terms \((t, \theta) = (y, \{y \mapsto 1 : y\})\) and \((t', \theta') = (y', \{y' \mapsto 1 : 1 : y'\})\). The cyclic term graph \((V, E, VL, EL)\) for \((t, \theta)\) is given by:

\[
V = \{1, 1 : y\}, \quad E = \{(1 : y, 1), (1 : y, 1 : y)\}
\]

\[VL(1) = \emptyset, \quad VL(1 : y) = \{y\}\]

\[EL(1 : y, 1) = 1, \quad EL(1 : y, 1 : y) = 2\]

The cyclic term graph \((V', E', VL', EL')\) for \((t', \theta')\) is given by:

\[
V' = \{1, 1 : y', 1 : 1 : y'\}, \quad E' = \{(1 : y', 1), (1 : y', 1 : 1 : y'), (1 : 1 : y', 1), (1 : 1 : y', 1 : y')\},
\]

\[VL'(1) = \emptyset, \quad VL'(1 : y') = \emptyset, \quad VL'(1 : 1 : y') = \{y'\}\]

\[EL'(1 : y', 1) = 1, \quad EL'(1 : 1 : y', 1) = 1\]

\[EL'(1 : 1 : y', 1 : y') = 2, \quad EL'(1 : 1 : y', 1 : y') = 2\]

The relation \(Z\), given below, is a bisimulation that witnesses that \(1 : y \in V\) and \(1 : 1 : y' \in V'\) bisimulate (and also witnesses that \(y\) and \(y'\) bisimulate).

\[Z = \{(1, 1), (1 : y, 1 : y'), (1 : y, 1 : 1 : y')\}\]

This bisimulation thus also witnesses that \((t, \theta) \equiv (t', \theta')\).

The two cyclic term graphs, together with the given bisimulation are drawn in Figure 1.

![Figure 1: Example of cyclic term graphs for \((y, \{y \mapsto 1 : y\})\) and \((y', \{y' \mapsto 1 : 1 : y'\})\), and a bisimulation. The bisimulation \(Z\) is drawn with a dashed line.](image-url)

The notion of equivalence of cyclic terms allows us to state the following property. This property will allow us to assume, without loss of generality, that cyclic terms are in base form.

Proposition 9. For every cyclic term there is an equivalent cyclic term in base form.
Proof. Let \((t, \theta)\) be a cyclic term. We show that \((t', \theta') = (\theta(t), \theta|_{FV\ar(\theta(t))})\) is an equivalent cyclic term in base form. Clearly, \((t', \theta')\) is a cyclic term.

We show that \((t', \theta')\) is equivalent to \((t, \theta)\). Consider the cyclic term graphs \((V, E, VL, EL)\) for \((t, \theta)\) and \((V', E', VL', EL')\) for \((t', \theta')\). Since \((t, \theta)\) and \((t', \theta')\) contain the same subterms (that are not frozen variables), we know \(V = V'\). Also, the construction of edges and edge labels in the cyclic term graph for a cyclic term \((t, \theta)\) is invariant under application of \(\theta\) to \(t\). Thus, both cyclic term graphs are isomorphic (when not taking into account vertex labels). Since the definition of bisimulation does not use vertex labels, we get that \(id\) witnesses that the term graphs bisimulate. Thus \((t, \theta)\) and \((t', \theta')\) are equivalent.

Now, we show that \((t', \theta')\) is in base form. Take an arbitrary \(x \in V\ar(rng(\theta'))\). Since \(\theta' \subseteq \theta\), we know \(x\) is in \(V\ar(rng(\theta))\) as well. Thus there is a \(y \mapsto t_y \in \theta\) such that \(x \in V\ar(t_y)\). By definition of cyclic terms, we know that \(y \in V\ar(t)\). Thus \(x \in V\ar(\theta(t)) = V\ar(t')\).

When dealing with terms containing variables, a notion of substitutions is required. We define substitutions on cyclic terms and their effect as follows.

Definition 10 (Substitutions). A substitution for cyclic terms is a mapping \(\sigma : \mathcal{X} \rightarrow \Sigma(\mathcal{X} \cup \mathcal{Y})\) that maps all but finitely many variables to themselves, together with a total mapping \(\omega : FV\ar(rng(\sigma)) \rightarrow \Sigma(\mathcal{X} \cup FV\ar(rng(\sigma))) \setminus V\ar(rng(\sigma))\). Substitutions are denoted with \((\sigma, \omega)\). We denote the unique homomorphic extension \(\check{\sigma}\) of \(\sigma\) on \(\Sigma(\mathcal{X} \cup \mathcal{Y})\) also with \(\check{\sigma}\) if no confusion arises. Similarly for the unique homomorphic extension \(\check{\omega}\) of \(\omega\) on \(\Sigma(\mathcal{X} \cup \mathcal{Y})\).

Definition 11 (Safe substitutions). A substitution \((\sigma, \omega)\) is safe for a cyclic term \((t, \theta)\) if \(dom(\omega) \cap V\ar(t) = \emptyset\). In other words, a substitution is safe for a cyclic term, if it doesn’t redefine (the mapping of) frozen variables that were already defined.

Definition 12 (Proper substitutions). A substitution \((\sigma, \omega)\) is proper for a cyclic term \((t, \theta)\) if \(V\ar(rng(\omega|_{V\ar(\sigma(t))})) \subseteq V\ar(\sigma(t))\). In other words, a substitution is proper for a cyclic term \((t, \theta)\), if the the mappings of all frozen variables introduced by instantiation of a variable in \(t\) contain no frozen variables that are not introduced by instantiation of a variable in \(t\).

Definition 13 (Effect of safe, proper substitutions). The result \((\sigma, \omega)(t, \theta)\) of a substitution \((\sigma, \omega)\), safe and proper for a cyclic term \((t, \theta)\), to this cyclic term \((t, \theta)\) is defined as \((\sigma(t), (\sigma(\theta) \cup \omega)|_{V\ar(\sigma(t))})\). Here \(\{y_1 \mapsto t_1, \ldots, y_n \mapsto t_n\}\) for some \(y_1, \ldots, y_n \mapsto \sigma(t_i)\).

Proposition 14. The result \((t', \theta')\) of a substitution \((\sigma, \omega)\), safe and proper for a cyclic term \((t, \theta)\), to this cyclic term \((t, \theta)\) is a cyclic term.

Proof. We show that the result \((t', \theta')\) is a cyclic term. Assume without loss of generality that \((t, \theta)\) is in base form. Since \((\sigma, \omega)\) is safe for \((t, \theta)\), we know that \(dom(\sigma(\theta)) \cap dom(\omega) = \emptyset\). Therefore \(\sigma(\theta) \cup \omega\) is a (functional) mapping, and thus so is \(t'\).

Clearly, \(dom(\theta') \subseteq dom(\theta')\). Take an arbitrary \(y \in V\ar(t')\). Then either (i) \(y \in V\ar(t)\) or (ii) for some \(x \in V\ar(t)\), we have \(x \mapsto t_y \in \sigma\) and some \(t_y \in V\ar(t_y)\) and \(y \in V\ar(t_y)\). In case (i), \(y \in V\ar(\sigma(t))\) and \(y \in dom(\theta)\). So also, \(y \in dom(\sigma(\theta))\). Therefore, \(y \in dom(\theta')\). In case (ii), \(y \in V\ar(\sigma(t))\) and \(y \in dom(\omega)\), thus \(y \in dom(\theta')\).

Finally, we show that \(FV\ar(\check{\theta'}(\sigma(\theta))) \subseteq V\ar(t')\). Take an arbitrary \(y \in FV\ar(\check{\theta'}(\sigma(\theta)))\). Then either (i) \(y \in V\ar(\check{\theta'}(\sigma(\theta)))\) or (ii) \(y \in V\ar(\check{\theta'}(\omega))\). In case (i), either (i.a) \(y \in V\ar(\check{\theta'}(\sigma(\theta)))\) or (i.b) for some \(x \in V\ar(\check{\theta'}(\sigma(\theta)))\), we have \(x \mapsto t_y \in \sigma\) and some \(t_y \in V\ar(t_y)\). In case (i.a), \(y \in V\ar(t)\), so \(y \in V\ar(t')\). In case (i.b), since \((t, \theta)\) is in base form, \(x \in V\ar(t)\) and thus \(y \in V\ar(t')\). In case (ii), we know that \(y \in V\ar(\check{\theta'}(\sigma(\theta)))\) and \(y \in FV\ar(\check{\theta'}(\sigma(\theta)))\). Then, since \((\sigma, \omega)\) is proper for \((t, \theta)\), we know that \(y \in V\ar(t')\).

We showed that \((t', \theta')\) has the property that \(\theta'\) is a total mapping from \(V\ar(t')\) to \(\Sigma(\mathcal{X} \cup V\ar(t'))\). Thus \((t', \theta')\) is a cyclic term. \(\square\)
These notions of equivalence and substitutions, defined above, allow us to define a notion of unification.

**Definition 15** (Unification). A substitution $(\sigma, \omega)$ unifies two cyclic terms $(t, \theta)$ and $(t', \theta')$ if

- it is safe and proper for both $(t, \theta)$ and $(t', \theta')$, and
- $(\sigma, \omega)(t, \theta) \equiv (\sigma, \omega)(t', \theta')$.

In order to avoid unwanted side-effects of operations on cyclic terms (or substitutions) that happen to share (frozen) variables, we define a method to standardize the frozen variables in two cyclic terms apart. We introduce frozen variable renamings, and use them to standardize the frozen variables in cyclic terms and substitutions apart.

**Definition 16** (Frozen variable renaming). A frozen variable renaming is a function $f : \mathcal{Y} \to \mathcal{Y}$ that maps finitely many $y \in \mathcal{Y}$ to different and pairwise disjoint $y' \in \mathcal{Y}$, and all other frozen variables to themselves. We denote the unique homomorphic extension $\hat{f}$ of a frozen variable renaming $f$ on $\Sigma(\mathcal{X} \cup \mathcal{Y})$ also with $f$, if no confusion arises.

**Definition 17** (Safe frozen variable renaming). A frozen variable renaming $f = \{y_1 \mapsto y'_1, \ldots, y_n \mapsto y'_n\}$ is safe for a cyclic term $(t, \theta)$ if for all $1 \leq i \leq n$ we have that $y'_i \notin \text{FVar}(t)$. A frozen variable renaming $f = \{y_1 \mapsto y'_1, \ldots, y_n \mapsto y'_n\}$ is safe for a substitution $(\sigma, \omega)$ if for all $1 \leq i \leq n$ we have that $y'_i \notin \text{dom}(\omega)$.

**Definition 18** (Renaming frozen variables in cyclic terms). The result $f(t, \theta)$ of a frozen variable renaming $f$, safe for a cyclic term $(t, \theta)$, to this cyclic term $(t, \theta)$ is the cyclic term $(f(t), f(\theta))$, where $f(\theta) = \{f(y) \mapsto f(t_y) \mid y \mapsto t_y \in \theta\}$. The fact that $f$ is safe for $(t, \theta)$, together with the fact that the range of $f$ is pairwise disjoint, assures that $f(t, \theta)$ is a cyclic term, and thus that applying a safe frozen variable renaming to a cyclic term results in a cyclic term.

**Observation 19.** For any cyclic term $(t, \theta)$ and any frozen variable renaming $f$, safe for $(t, \theta)$, we have that $(t, \theta) \equiv f(t, \theta)$.

**Proof (sketch).** There exists a trivial bisimulation between the cyclic term graphs of $(t, \theta)$ and $f(t, \theta)$.

**Definition 20** (Renaming frozen variables in substitutions). The result $f(\sigma, \omega)$ of a frozen variable renaming $f$ to the substitution $(\sigma, \omega)$ is the substitution $(f(\sigma), f(\omega))$. The fact that $f$ is safe for $(\sigma, \omega)$, together with the fact that the range of $f$ is pairwise disjoint, ensures that the result is a substitution.

**Proposition 21.** Let $(t, \theta)$ be a cyclic term, and let $(\sigma, \omega)$ be a substitution, safe and proper for $(t, \theta)$. Also, let $f$ be a frozen variable renaming safe for $(\sigma, \omega)$. If $f(\sigma, \omega)$ is safe and proper for $(t, \theta)$, then we have that $(f(\sigma, \omega))(t, \theta) \equiv (\sigma, \omega)(t, \theta)$.

**Proof (sketch).** Since $f$ has an effect only on frozen variables, and since the construction of cyclic term graphs is invariant under (uniform) substitution of frozen variables, the cyclic term graphs for $(f(\sigma, \omega))(t, \theta)$ and $(\sigma, \omega)(t, \theta)$ are identical. Thus $\text{id}$ is the bisimulation that witnesses that $(f(\sigma, \omega))(t, \theta) \equiv (\sigma, \omega)(t, \theta)$.

**Definition 22** (Standardization apart). Two cyclic terms $(t, \theta)$ and $(t', \theta')$ are said to have their frozen variables standardized apart if $\text{FVar}(t) \cap \text{FVar}(t') = \emptyset$. Two substitutions $(\sigma, \omega)$ and $(\sigma', \omega')$ are said to have their frozen variables standardized apart if $\text{dom}(\omega) \cap \text{dom}(\omega') = \emptyset$.  

6
Cyclic terms and substitutions for cyclic terms can be standardized apart by means of safe variable renaming. In the following, we will assume that cyclic terms and substitutions have their frozen variables standardized apart. With this method of standardization apart in place, we can return to substitutions and define composition of substitutions (that have their frozen variables standardized apart).

**Definition 23** (Composition of substitutions). Let $(\sigma, \omega)$ and $(\sigma', \omega')$ be two substitutions, with frozen variables standardized apart. We define the composition of these two substitutions to be $(\sigma, \omega) \circ (\sigma', \omega') = (\sigma \circ \sigma', \omega \cup \omega')$. The fact that the two substitutions have their frozen variables standardized apart ensures that the result is a substitution.

**Proposition 24.** Let $(\sigma, \omega)$ and $(\sigma', \omega')$ be two substitutions, that have their frozen variables standardized apart. Let $(\sigma', \omega')$ be safe and proper for a given cyclic term $(t, \theta)$, and let $(\sigma, \omega)$ be safe and proper for $(\sigma', \omega')(t, \theta)$. Then $(\sigma, \omega) \circ (\sigma', \omega')$ is safe and proper for $(t, \theta)$.

**Proof.** Let $(\sigma'', \omega'') = (\sigma, \omega) \circ (\sigma', \omega') = (\sigma \circ \sigma', \omega \cup \omega')$. Since $(\sigma, \omega)$ and $(\sigma', \omega')$ have their frozen variables standardized apart, we know $(\sigma'', \omega'')$ is a substitution.

We show that $(\sigma'', \omega'')$ is safe for $(t, \theta)$. Since $(\sigma', \omega')$ is safe for $(t, \theta)$, we know that $\text{dom}(\omega') \cap \text{FVar}(t) = \emptyset$. Also, since $(\sigma, \omega)$ is safe for $(\sigma', \omega')(t, \theta)$, we know that $\text{dom}(\omega \cup \omega') \cap \text{FVar}(t) = \emptyset$. This implies that $\text{dom}(\omega) \cap \text{FVar}(t) = \emptyset$. Thus $(\text{dom}(\omega \cup \omega')) \cap \text{FVar}(t) = \emptyset$. And thus $\text{dom}(\omega \cup \omega') \cap \text{FVar}(t) = \emptyset$. Thus $(\sigma'', \omega'')$ is safe for $(t, \theta)$.

We show that $(\sigma'', \omega'')$ is proper for $(t, \theta)$. Take an arbitrary $y \in \text{FVar}(\text{rng}(\omega''|_{\text{FVar}(\sigma''(t))}))$. We show that $y \in \text{FVar}(\sigma''(t))$. We know that there exists a $y' \mapsto t_y \in \omega''$ such that $y' \in \text{FVar}(\sigma''(t))$ and $y \in \text{FVar}(t_y)$. Since $y' \in \text{FVar}(\sigma'(t))$, we know that either (i) $y' \in \text{FVar}(\sigma'(t))$ or (ii) for some $x \in \text{Var}(\sigma'(t))$ we have that $y' \in \text{FVar}(\sigma'(x))$. In case (i), since $(\sigma', \omega')$ is proper for $(t, \theta)$, we know that $y \in \text{FVar}(\sigma'(t))$, and thus also $y \in \text{FVar}(\sigma''(t))$. In case (ii), since $(\sigma, \omega)$ is proper for $(\sigma', \omega')(t, \theta)$, we know $y \in \text{FVar}(\sigma'(t)) = \text{FVar}(\sigma''(t))$. \square

Finally, we introduce some notational conventions. Firstly, we will often denote cyclic terms $(t, \theta)$ with $t$, and denote substitutions for cyclic terms $(\sigma, \omega)$ with $\sigma$. Secondly, we introduce a notation to refer to subterms of cyclic terms. In order to do so, we define the notion decomposable form for cyclic terms.

**Definition 25.** A cyclic term $(t, \theta)$ is in decomposable form if $t \notin \mathcal{V}$ and for every direct subterm $t' \in \text{sub}(t)$ of $t$ holds that $\text{FVar}(\text{rng}(\theta|_{\text{FVar}(t')})) \subseteq \text{FVar}(t')$.

**Proposition 26.** For every cyclic term $(t, \theta)$, there is an equivalent cyclic term $(t', \theta')$ in decomposable form.

**Proof (sketch).** For any cyclic term $(t, \theta)$ and any mapping $y \mapsto t_y \in \theta$, applying the mapping $\rho = \{ y \mapsto t_y \}$ to any subterm of $t$ or $\text{rng}(\theta)$ results in a cyclic term $(t', \theta')$ equivalent to $(t, \theta)$ (after restricting the domain of $\theta'$ if necessary). By applying a finite number of such equivalence preserving transformations, any cyclic term can be transformed into decomposable form. \square

In the following, we assume without loss of generality that cyclic terms are in decomposable form. Any cyclic terms that are not in decomposable form, we implicitly transform to decomposable form.

**Definition 27.** We say that a cyclic term $(t, \theta)$ is of the form $d(t_1, \ldots, t_n)$, for $d \in \mathcal{C} \cup \mathcal{F}$, if $t = d(t_1, \ldots, t_n)$. We assume without loss of generality that $(t, \theta)$ is in decomposable form. In this case, for $1 \leq i \leq n$, we let $(t_i, \theta_i)$ denote the cyclic term that is obtained by letting $\theta_i = \theta|_{\text{FVar}(t_i)}$. 

7
Thirdly, we define substitution of subterms of cyclic terms.

**Definition 28.** For a given cyclic term \((t, \theta)\) we define the (possibly infinite) set of positions \(\text{pos}(t, \theta) \subseteq \mathbb{N}^*\) as follows.

\[
\text{pos}(x, \theta) = \{\epsilon\}
\]

\[
\text{pos}(d(t_1, \ldots, t_n), \theta) = \{\epsilon\} \cup \{1 \cdot \overline{m} \mid \overline{m} \in \text{pos}(t_1)\} \cup \cdots \cup \{n \cdot \overline{m} \mid \overline{m} \in \text{pos}(t_n)\}
\]

Note that we do not explicitly handle the case for \(t \in \mathcal{Y}\), since we implicitly transform cyclic terms to decomposable form.

**Definition 29** (Fixed-position substitutions). For given cyclic terms \((t, \theta)\) and \((t', \theta')\) (with their frozen variables standardized apart), and a position \(\overline{m} \in \text{pos}(t, \theta)\) of \((t, \theta)\), we define the fixed-position substitution \((t, \theta)[(t', \theta')]_{\overline{m}}\) as:

\[
(t, \theta)[(t', \theta')]_{\overline{m}} = (t[t'[\overline{m}], (\theta \cup \theta')|_{t[t'[\overline{m}]}})
\]

\[
d(t_1, \ldots, t_n)[t'[\overline{m}] = d(t_1, \ldots, t_i-1, t_i[t'[\overline{m}], t_{i+1}, \ldots, t_n)
\]

Again, we do not explicitly handle the case for \(t \in \mathcal{Y}\), since we implicitly transform cyclic terms to decomposable form. Note that since the two cyclic terms \((t, \theta)\) and \((t', \theta')\) are in decomposable form and have their frozen variables standardized apart, the result \((t, \theta)[(t', \theta')]_{\overline{m}}\) of a fixed-position substitution is a cyclic term.

Fourthly, in the following, we will denote cyclic terms \((t, \theta)\) where \(\theta = \{y_1 \mapsto t_1, \ldots, y_n \mapsto t_n\}\) in Curry notation as \(t \{y_1 \rightarrow t_1, \ldots, y_n \rightarrow t_n\}\).

Note that in this paper, we will not make the connection between cyclic terms in the above syntactic sense and regular terms precise. The intuition is that the set of cyclic terms corresponds to the set of (finitely-branching) regular terms. Making this intuition precise is a topic of further research, and is closely related to the topic of finding a declarative semantics of our approach (see Section 5).

### 2.1.1 Adding typing

For the purposes of functional logic programming, we would like to extend the above machinery with the concept of types. For the sake of simplicity of presentation, we have not done this explicitly above. Extending the above with typing is, however, very straightforward. We describe the steps that have to be taken to extend the above with typing.

We need a many sorted signature \(\Sigma\), to start with. Also, we explicitly partition the set \(\mathcal{X}\) of variables and the set \(\mathcal{Y}\) of frozen variables into sets for each sort. Then, for cyclic terms \((t, \theta)\), we pose the usual typing constraints on \(t\) and on all mappings in \(\theta\). For substitutions \((\sigma, \omega)\) on cyclic terms, we pose the usual typing constraints on all mappings in \(\sigma\) and \(\omega\). Finally, we straightforwardly add typing constraints on frozen variable renamings and fixed-position substitutions.

### 2.1.2 Computing unification

An algorithm to compute whether a unifier for two cyclic terms (that have their frozen variables standardized apart) exists, and to compute such a unifier, is given in Algorithm 1. This algorithm is based on the Martelli-Montanari algorithm [6].

**Theorem 30** (Termination). The unification algorithm always terminates, no matter what choices are made.
Proof (sketch). Similarly to the proof of termination for the Martelli-Montanari algorithm, we define a well-founded ordering that decreases with every action taken. To the ordering used in [6], we add (as most significant dimension), the following measure.

$$\text{maxsize}(H) - \text{size}(H)$$

We know $\theta$ and $\theta'$ are fixed and finite. Since only finitely many variables from $\mathcal{X}$ occur in $E$ (and the number of such variables never increases), the number of equations in $F$ is finitely bounded. For every substitution $x \mapsto t$ applied to terms in $E$, the size of $t$ is bounded by the size of $E$ (at the time of applying $x \mapsto t$). For every substitution $y \mapsto t \in F$, the size of $t$ is also bounded by the size of $E$ (at the time of adding $y \mapsto t$ to $F$). For every substitution $y \mapsto t \in \theta \cup \theta'$, the size of $t$ is bounded as well. There are finitely many different possible mappings that can be applied to equations in $E$. Furthermore, there exists an upper bound on this number of mappings. Thus, since $E$ is initially finite, we know there is an upper bound for the number of different equations that can occur in $E$. Since only equations occurring in $E$ can be added to $H$, the size of $H$ also has this bound. Thus $\text{maxsize}(H)$ is well-defined, and therefore this is a valid measure.

It is easy to see that with each action, this well-founded ordering decreases. Thus the algorithm terminates.

Theorem 31 (Partial correctness). On termination of the algorithm, a unifying substitution is returned iff a unifying substitution exists.

Proof (sketch). It is easy to see that if an $(E, F)$ is returned, then it is a substitution. If $E$ contains anything other than equations of the form $x \mapsto t$, where $x$ occurs only once in $E$, then there are still actions that can be performed. Also, for any mapping $y \mapsto t \in F$, we know that $y$ occurs in the rhs of some mapping in $E$, because of the structure of the only rule introducing mappings to $F$. Because of the structure of this same rule, we have that for any $y$ that occurs in the rhs of a mapping in $E$, there is a mapping $y \mapsto t \in F$. Furthermore, because of the structure of this same rule, we know that $(E, F)$ is safe and proper for both $(t, \theta)$ and $(t', \theta')$.

In order to see that $(E, F)$ is a unifier of $(t, \theta)$ and $(t', \theta')$, consider the following condition, in which we fix suitable mappings $\sigma$ and $\omega$.

There exists a bisimulation between the cyclic term graphs

$$(\sigma(t), (\theta \cup \theta' \cup \omega)\sigma(t)) \text{ and } (\sigma(t'), (\theta \cup \theta' \cup \omega)\sigma(t'))$$

for each $t \equiv t' \in E \cup H$.

This condition holds on successful termination of the algorithm, for $(\sigma, \omega) = (E, F)$. For all equations $x \equiv t_x \in E$, this holds since $x \mapsto t_x \in \sigma$. For equations $y \equiv t_y \in H$ the condition holds as well. We know $y \equiv t_y$ can only be in $H$ if it was in $E$ before. If there is no such bisimulation (as specified in the condition), by the structure of the rules, the algorithm would return with $\bot$. However, since the algorithm terminated successfully (by assumption), we know such a bisimulation must exist.

We know that the algorithm can only halt with failure because of rule (2). By the structure of rule (2), we know if the algorithm halts with failure, the condition does not hold (for any $\sigma$). It suffices to show, for each (other) action that can be performed, that the condition holds before applying the action if it holds after applying the action. The cases for rules (1), (3), (4) and (5) are trivial. The cases for rules (6), (7), (8) and (9) can be proven straightforwardly. This proves that the condition holds on initiating the algorithm, for $(\sigma, \omega) = (E, F)$, if the algorithm halts with $(E, F)$. It also proves that the condition holds for no $(\sigma, \omega)$ on initiating the algorithm if the algorithm halts with $\bot$. From this claim, it follows that the algorithm halts with a unifier $(E, F)$ iff such a unifier exists.
Algorithm 1: Unification algorithm for cyclic terms.

UNIFY((t, θ), (t′, θ′))

Input: Two cyclic terms (t, θ) and (t′, θ′), with frozen variables standardized apart.

Output: A unifier (σ, ω) of the two cyclic terms if it exists, or else ⊥.

Let E = {t = t′}. Let H = ∅. Let F = ∅. While some action can be performed, and no failure has occurred, do the following. Nondeterministically choose from the set E of equations an equation of a form below (such that the corresponding condition holds) and perform the corresponding action. Note that x ranges over variables in X, y over frozen variables in Y, and t over terms in Σ(X ∪ Y).

| form | condition | action |
|------|-----------|--------|
| 1) f(s₁,...,sₙ) ≡ f(t₁,...,tₙ) | t | replace by the equations s₁ = t₁, ... , sₙ = tₙ |
| 2) f(s₁,...,sₙ) ≡ g(t₁,...,tₙ) | f ≠ g | halt with failure |
| 3) t ≡ x | ⊤ | delete the equation |
| 4) t ≡ x | t ∉ X | replace by the equation x ≡ t |
| 5) t ≡ y | t ∉ X ∪ Y | replace by the equation y ≡ t |
| 6) x ≡ t | x ∉ Var(t) and x occurs in another equation | perform the substitution x → t on all other equations |
| 7) x ≡ t | x ∈ Var(t) | for a fresh y: replace by the equation x ≡ y; apply the substitution x → y, on all other equations; and add y → t(x → y) to F |
| 8) y ≡ t | y ≡ t ∈ H | delete the equation |
| 9) y ≡ t | y ≡ t ∉ H | add y ≡ t to H; and apply the unique substitution y → t from θ ∪ θ′ ∪ F to t |

When the algorithm terminates with failure, return ⊥. Otherwise, return (E, F), where E is seen as a mapping from X to Σ(X ∪ Y).

2.2 Systems of equations

Another possible representation of regular (infinite) terms would be systems of equations. Such representations of infinite terms can, for instance, be found in (certain implementations of) Prolog and several languages used for coinductive specifications. Because of their use in different contexts, the reader might be familiar with systems of equations, and might appreciate a motivation for the choice of using cyclic terms instead of systems of equations. In fact, there is a one-to-one correspondence between cyclic terms and systems of equations. Below, we illustrate how cyclic terms can be straightforwardly transformed to systems of equations, and vice versa. A major advantage of the use of cyclic terms is that this manner of representing regular terms allows us to to define the operational semantics in such a way that the connection to the operational semantics of regular Curry is very clear (see Section 3).

2.2.1 Correspondence with cyclic terms

We informally describe how systems of equations can be transformed straightforwardly to cyclic terms, and vice versa. We will do so by means of an example. A formal definition of systems of
equations and the transformations to and from cyclic terms is beyond the scope of this paper.

Example 32. Consider the regular infinite tree represented by the cyclic term, for symbols $f, g, h \in \Sigma$:

$$(f(g(y, y')), \{y \mapsto g(y, y'), y' \mapsto h(y')\})$$

This regular tree can also be represented by the following set of equations, with root $x$:

$$\{ x = f(g(x', x''), \quad x' = g(x', x''), \quad x'' = h(x'') \}$$

It is easy to see that a cyclic term $(t, \theta)$ can be written as a system of equations by mapping each frozen variable to a (regular) variable, and (possibly) adding an extra variable as root to refer to $t$.

Conversely, a (finite) system of equations can be written as a cyclic term $(t, \theta)$ as follows. We map the variables that occur on the left hand side of equations to (distinct) frozen variables. We take the frozen variable corresponding to the distinguished variable in the equation as $t$. We let $\theta$ be the mappings corresponding to the equations. Possibly, we need to unfold $t$ by applying mappings from $\theta$ until all frozen variables occurring in $\theta$ also occur in $t$.

3 Computational strategy

With the (well-behaved) data structures to represent regular terms in place, we are ready to define the computational strategy that uses generalized circular coinduction. Note that in the following, we use cyclic terms (often simply denoted terms) and the corresponding notions of substitutions and unifiers.

3.1 Informally

The intuition behind our computational strategy is fairly simple. The idea is to detect circularity in the reductions performed, and breaking out of such a circle by assigning a possible value to the (function symbol rooted) term which is evaluated repeatedly. What the possible values are is something that needs to be specified by the programmer. The reason for this is discussed in Section 3.3.

More concretely, this results in the following computational mechanism, based on the needed narrowing mechanisms from [1, 7]. Analogously to the method of detecting cyclic behavior in [8], in every derivation, we keep track of the (function symbol rooted) expressions on which narrowing is applied in the derivation, together with the total term they appear in and their position in this total term. As soon as the term $t$ (appearing as subterm of the total term $t_{tot}$) on which narrowing is about to be applied, and some term $t'$ (appearing as subterm of the total term $t'_{tot}$) on which narrowing is applied previously, unify (with unifier $\sigma$), we do the following.

- We (nondeterministically) guess a possible resulting value for the function symbol rooted expression occurring at the root of both $t$ and $t'$, from the possibilities specified by the programmer.

- To the derivation, we add the constraint $\sigma(t'_{tot}) \doteq \sigma(t_{tot}')$, where $t'_{tot}$ is the term $t_{tot}$ in which the subterm $t$ is replaced by the guessed result, and $t_{tot}'$ is defined similarly.

- We continue the derivation with $\sigma(t'_{tot})$.

Also, if during the derivations we encounter constraints of the form $t \doteq t'$, we try to unify them.
3.2 Formal definition

In order to formally define our computational mechanism, we recall some preliminary notions, known from the literature on functional logic programming and Curry. We define patterns and (partial) definitional trees.

**Definition 33** (Taken from Definition 2 in [7]). A pattern is a term \( f(t_1, \ldots, t_n) \), where \( n \) is the arity of \( f \in \mathcal{F} \) and each \( t_i \), for \( 1 \leq i \leq n \), is a constructor term.

**Definition 34** (Definition 12 in [7]). \( \mathcal{T} \) is a partial definitional tree (pdt) with pattern \( \pi \) iff one of the following cases holds:

- \( \mathcal{T} = \text{branch}(\pi, o, \mathcal{T}_1, \ldots, \mathcal{T}_k) \), where \( \pi \) is a pattern, \( o \) is the occurrence of a variable in \( \pi \), the sort of \( \pi|_o \) has constructors \( c_1, \ldots, c_k \) for some \( k > 0 \), and for all \( i \in \{1, \ldots, k\} \), \( \mathcal{T}_i \) is a pdt with pattern \( \pi|_{c_i[X_1, \ldots, X_n]}|_o \), where \( n \) is the arity of \( c_i \) and \( X_1, \ldots, X_n \) are new distinct variables.

- \( \mathcal{T} = \text{leaf}(\pi) \), where \( \pi \) is a pattern.

**Definition 35** (Taken from \([7]\) and \([1]\)). Let \( \mathcal{R} \) be a rewrite system. \( \mathcal{T} \) is a definitional tree of an operation \( f \) iff \( \mathcal{T} \) is a pdt whose pattern argument is \( f(X_1, \ldots, X_n) \), where \( n \) is the arity of \( f \) and \( X_1, \ldots, X_n \) are new distinct variables, and for every rule \( l \to r \) of \( \mathcal{R} \) with \( l = f(t_1, \ldots, t_n) \), there exists a leaf \( \text{leaf}(\pi) \) of \( \mathcal{T} \) such that \( l \) is a variant of \( \pi \), and we say that the node \( \text{leaf}(\pi) \) represents the rule \( l \to r \), in which case we will also write \( \text{leaf}(l \to r) \). We write \( \text{pattern}(\mathcal{T}) \) for the pattern of a definitional tree \( \mathcal{T} \) and \( \mathcal{DT} \) for the set of all definitional trees.

Besides the notions used in the usual computational mechanism of Curry, we will need a few additional notions to make our computational strategy work. We define memory configurations, and several types of expressions used in the computation.

**Definition 36.** We denote with \( \text{Mem} \) the set of all memory configurations \( \mathcal{P}(\mathcal{CT} \times \mathcal{CT} \times \mathbb{N}^*) \). A memory configuration thus is a set of triples consisting of two cyclic terms and a sequence of natural numbers (representing a position in the second cyclic term).

**Definition 37.** We denote with \( \text{EQ} \) the set of all equations of the form \( t \doteq t' \), where \( t, t' \in \mathcal{CT} \), and the trivial (dummy) equation \( \top \).

**Definition 38** (Based on definitions in \([1]\)). An answer expression is a pair \( \langle \sigma, e \rangle \), consisting of a substitution \( \sigma \) and an expression \( e \). An answer expression \( \langle \sigma, e \rangle \) is solved if \( e \) is a constructor term. A disjunctive expression is a (multi-)set of answer expressions \( \{\langle \sigma_1, e_1 \rangle, \ldots, \langle \sigma_n, e_n \rangle\} \), sometimes written as \( \langle \sigma_1, e_1 \rangle \lor \ldots \lor \langle \sigma_n, e_n \rangle \). The set of all disjunctive expressions is denoted by \( \mathcal{D} \). A memory answer expression is a pair \( \langle \sigma, e, M \rangle \), where \( \langle \sigma, e \rangle \) is an answer expression, and \( M \in \text{Mem} \) is a memory configuration. A memory answer expression \( \langle \sigma, e, M \rangle \) is solved if the answer expression \( \langle \sigma, e \rangle \) is solved. A disjunctive memory expression is a (multi-)set of memory answer expressions. Notation is analogous to the notation of disjunctive expressions.

The set of all disjunctive memory expressions is denoted \( \mathcal{D}_{\text{Mem}} \). A memory constraint answer expression is a pair \( \langle \sigma, e, M, eq \rangle \), where \( \langle \sigma, e, M \rangle \) is a memory answer expression, and \( eq \in \text{EQ} \) is an equation. A disjunctive memory constraint expression is a (multi-)set of memory constraint answer expressions. Notation is analogous to the notation of disjunctive expressions.

In order to handle the additional types of expressions defined above, we adapt several auxiliary functions known from the usual computational strategy of Curry.
Definition 39. We define the function \( \text{pair} : (\mathcal{D}_{\text{Mem}} \cup \{\text{suspend}\}) \times \text{EQ} \to \mathcal{D}_{\text{Mem}, \text{EQ}} \cup \{\text{suspend}\} \) as follows.

\[
\text{pair}(\text{suspend}, eq) = \text{suspend}
\]

\[
\text{pair}(\{(\sigma_1, e_1, M_1), \ldots, (\sigma_n, e_n, M_n)\}, eq) = \{(\sigma_1, e_1, M_1, eq), \ldots, (\sigma_n, e_n, M_n, eq)\}
\]

Definition 40 (Based on definitions in [1]). We define the auxiliary functions \( \text{compose} \) and \( \text{replace} \) as follows. Note that the function \( \text{cst} \) will be defined below.

\[
\text{compose}(t, T, \sigma, t', o', M) = \begin{cases} 
\{(\sigma, t, M')\} & \text{if } \text{cst}(t, T, t', o', M) = \text{suspend} \\
\{(\sigma_1 \circ \sigma, t_1, M_1), \ldots, (\sigma_n \circ \sigma, t_n, M_n)\} & \text{if } \text{cst}(t, T, t', o', M) = \{(\sigma_1, t_1, M_1), \ldots, (\sigma_n, t_n, M_n)\}
\end{cases}
\]

\[
\text{replace}(t, o, \text{suspend}) = \text{suspend}
\]

\[
\text{replace}(t, o, \{(\sigma_1, t_1, M_1, eq_1), \ldots, (\sigma_n, t_n, M_n, eq_n)\}) = \{(\sigma_1, t_1, M_1, eq_1), \ldots, (\sigma_n, t_n, M_n, eq_n)\}
\]

With all the preliminary notions in place, we can now define formally what possible guesses for circular behaviour are.

Definition 41. Possible guesses are defined by a partial function \( \rho : CT \to \mathcal{P}(CT) \). We say that \( t' \) is a possible guess for a cyclic term \( t \in CT \) if \( t \in \text{dom}(\rho) \) and \( t' \in \rho(t) \).

We are now ready to define the two functions \( \text{cs} \) and \( \text{cst} \) that handle the computational steps in derivations.

Definition 42 (Based on definitions in [1]). The function \( \text{cs} : CT \times CT \times \mathbb{N}^* \times \text{Mem} \to \mathcal{D}_{\text{Mem}, \text{EQ}} \cup \{\text{suspend}\} \), designed to perform a computation step, is defined as follows.

\[
\text{cs}(x, t_{tot}, o, M) = \text{suspend} \quad \text{for all variables } x \in \mathcal{X} \quad (1)
\]

\[
\text{cs}(t \neq t', t_{tot}, o, M) = \{(\sigma, T, M, \top)\} \quad \text{if } \sigma \text{ unifies } t \text{ and } t' \quad (2)
\]

\[
\text{cs}(f(t_1, \ldots, t_n), t_{tot}, o, M) = \text{pair}(\text{cst}(f(t_1, \ldots, t_n), T, t_{tot}, o, M), \top) \quad \text{if } T \text{ is a definitional tree for } f \text{ and rule (2) does not apply} \quad (3.1)
\]

\[
\cup
\]

\[
\{(\sigma, g, M, \sigma(t_{tot})[g]_o \overset{\sigma(t'_{tot})[g]_{o'}}{=} \sigma(t'_{tot})[g]_{o'}) | (t', t'_{tot}, o') \in M, \sigma \text{ unifies } t \text{ and } t', g \text{ is a possible guess for } \sigma(t)\} \quad \text{where } t = f(t_1, \ldots, t_n) \quad (3.2)
\]

\[
\text{cs}(c(t_1, \ldots, t_n), t_{tot}, o, M) = \text{replace}(c(t_1, \ldots, t_n), k, \text{cs}(t_k, t_{tot}, o \cdot k, M)) \quad \text{if } \text{cs}(t_i, t_{tot}, o \cdot i, M) = \text{suspend}, \text{ for all } 1 \leq i < k \text{ and } \text{cs}(t_k, t_{tot}, o \cdot k, M) \neq \text{suspend} \quad (4)
\]

\[
\text{cs}(c(t_1, \ldots, t_n), t_{tot}, o, M) = \text{suspend} \quad \text{if } \text{cs}(t_i, t_{tot}, o \cdot i, M) = \text{suspend} \text{ for all } 1 \leq i \leq n \quad (5)
\]
Notice the use of nondeterminism in Definition 42 in particular in case (3). In order to allow
the programmer to use any form of nondeterminism, we opt for as much nondeterminism in
the computational strategy as possible. The choice to break cyclicity in derivations (case (3.2)) or
simply continue the derivation (case (3.1)) is nondeterministic. When breaking cyclicity, only
the single currently evaluated term is replaced by a possible guessed value, not all occurrences
of this term in the answer expression. Also, we do not remember what guessed values we have
applied in a derivation. This way, different guesses can be applied to (several occurrences of) the
same term in one derivation.

**Definition 43** (Based on definitions in [1]). The function \( cst : CT \times DT \times CT \times \mathbb{N}^* \times Mem \rightarrow (\mathcal{D}_{Mem} \cup \{\text{suspend}\}) \), designed to perform the application of a reduction (narrowing) step, is
deﬁned as follows.

\[
\begin{align*}
\text{cst}(t, \text{leaf}(l \rightarrow r), t', o', M) &= \{(id, \sigma(r), M \cup \{(t, t', o')\})\} \quad \text{if } \sigma \text{ is a substitution with } \sigma(l) = t \\
\text{cst}(t, \text{branch}(\pi, o, T_1, \ldots, T_k), t', o', M) &= \left\{ \begin{array}{ll}
\text{cst}(t, T_i, t', o', M) & \text{if } t|_o = c(t_1, \ldots, t_n) \text{ and } \\
& \text{pattern}(T_i)|_o = c(X_1, \ldots, X_n) \\
\emptyset & \text{if } t|_o = c(t_1, \ldots, t_n) \text{ and } \\
& \text{pattern}(T_i)|_o \neq c(\ldots), i = 1, \ldots, k \\
\bigcup_{i=1}^{k} \text{compose}(\sigma_i(t), T_i, \sigma_i, t', o', M) & \text{if } t|_o = X \text{ and } \\
& \sigma_i = \{X \mapsto \text{pattern}(T_i)|_o\} \\
\text{replace}(t, o, \text{cs}(t|_o, t', o'), M) & \text{if } t|_o = f(t_1, \ldots, t_n)
\end{array} \right.
\]

The main difference between the above definitions for the functions \( cs \) and \( cst \) and their
definitions in [1] is that in the above definitions we keep track of (1) a memory configuration \( M \),
(2) the total term \( t_{\text{tot}} \) (resp. \( t' \)) on which the computation step takes place and (3) the position
\( o \) (resp. \( o' \)) in this total term that is currently being evaluated. Also, with each application of a
reduction rule (the first case for \( cst \)), we add an appropriate entry to the memory configuration.
This allows us to resolve the circular behavior in case (3.2) for \( cs \), exactly as described in
Section 3.1.

We remember each term on which a reduction step is applied (the first case for \( cst \)). It would
be sufficient to remember only those terms for which a possible guess is available. This would
result in an equivalent (yet in principle more efficient) computational mechanism. For the sake
of simplicity of presentation, we don’t make this distinction.

Finally, we introduce a few standard reduction rules and we define what derivations are. We
use the following definitions of \( \land \) and \( \Rightarrow \), as in [1].

\[
\begin{align*}
\top \land X & \Rightarrow X \\
X \land \top & \Rightarrow X \\
(\top \Rightarrow X) & \Rightarrow X
\end{align*}
\]

**Definition 44** (Based on definitions in [1]). A rewriting step for a disjunctive memory expression
(denoted with \( \Rightarrow \)) goes as follows.

\[
D \lor \langle \sigma, e, M \rangle \lor D' \Rightarrow D \lor \langle \sigma \circ \sigma, eq_1 \Rightarrow e_1, M_1 \rangle \lor \ldots \lor \langle \sigma_n \circ \sigma, eq_n \Rightarrow e_n, M_n \rangle \lor D'
\]

if \( \langle \sigma, e, M \rangle \) is unsolved and \( \text{cs}(e, e, M) = \{(\sigma_1, e_1, M_1, eq_1), \ldots, (\sigma_n, e_n, M_n, eq_n)\} \)

Here, we let \( \top \Rightarrow t \) denote \( t \). A derivation is a finite sequence of rewriting steps. A derivation
for a term \( e \) is a derivation starting with the disjunctive memory expression \( \langle \emptyset, e, \emptyset \rangle \).
3.3 Restrictions on the assumptions

The need to specify explicitly what are possible resulting values for a term that is evaluated repeatedly in a circular derivation, is illustrated nicely by the following example. Consider the following program, given in Curry syntax.

\[
\text{and} \, [\,] = \text{True} \\
\text{and} \, (x:xs) = x \land \text{and} \, xs \\
\text{trues} = \text{True}:\text{trues}
\]

Without any restrictions on the resulting values that can be guessed in circular derivations, we would be able to derive both \text{True} and \text{False} from \text{and trues} (by guessing \text{True} and \text{False}, respectively, as the resulting value of \text{and trues}). However, if we want to interpret \text{and} as conjunction over (finite and infinite) lists of Boolean values, only the answer \text{True} would be appropriate. Also, if we want to interpret \text{and} as the predicate denoting all finite lists of Boolean values whose conjunction is true, only the answer \text{False} would be appropriate. Since the interpretation is to be determined by the programmer, we need to let the programmer specify such choices. The possible guesses can intuitively be considered as the possible meaning of cyclic behavior.

An advantage of placing explicit restrictions on the possible values for such nondeterministic steps in the derivation, is that the number of possibilities is reduced enormously. This obviously has a positive effect on the efficiency of computation.

How to specify the possible resulting values for each situation is a question that needs to be investigated further. With a program, the user needs to specify possible guesses with a partial function \( \rho : CT \rightarrow \mathcal{P}(CT) \). A concise specification language in which every (reasonable) choice of values can be expressed is desirable. In this paper we will describe the possible guesses with rewrite rules of the form \( f \Rightarrow v \) (where \( f \) is a term, and \( v \) a set of terms), that get the following interpretation. If a term unifies with \( f \) with unifier \( \theta \), the possible guesses of the value of \( f \) are the values in the set \( \theta(v) \). When \( v \) contains the anonymous variable \( _{\;} \), by convention, we let the corresponding possible guess \( \theta(_{\;}) \) be a fresh variable.

4 Examples

In order to illustrate the merit of our approach, we will discuss several examples. These examples show that certain problems can be programmed and solved in a very intuitive fashion in our approach. We will also argue that the problems exemplified here cannot be solved in a similarly intuitive manner in regular Curry. A number of derivations for the following examples can be found in Appendix A.

4.1 Infinite lists

In the program \( P_1 \) in Figure 2 we consider infinite lists. This example shows that it is possible to use the usual definitions of conjunction (\text{and}), quantifiers (\text{forall} and \text{exists}) and concatenation (\text{append}) to extend these functions to (regular) infinite lists, with the intended interpretation. Also, we can straightforwardly introduce a function \text{inf} that decides whether infinitely many elements of a list satisfy a certain property.

We can make the following derivations by using program \( P_1 \). These example derivations illustrate that the operations \text{and, forall, exists, inf} and \text{append} get their intended interpretation on infinite lists.
Reduction rules:

and [] = True
and (x:xs) = x && (and xs)

map f [] = []
map f (x:xs) = (f x):(map f xs)

forall f xs = and (map f xs)
exists f xs = not (and (map ((\x -> not (f x))) xs))

inf f xs = inf' f False (loop xs)
inf' f b (x:xs) = inf' f ((f x) || b) xs

loop (x:xs) = loop xs

append [] ys = ys
append (x:xs) ys = x:(append xs ys)

isEven n = mod n 2 == 0
isOdd n = not (isEven n)

ones = 1:ones
twos = 2:twos

Assumption possibilities:
and _  \rightarrow \{True\}
inf’ b _  \rightarrow \{b\}
loop x _  \rightarrow \{x\}
append xs _  \rightarrow \{_\}

Figure 2: Example program \( P_1 \).

| term               | value | substitution |
|--------------------|-------|--------------|
| forall isOdd ones  | \rightarrow* | True         |
| exists isOdd twos  | \rightarrow* | False        |
| inf isOdd (2:ones) | \rightarrow* | True         |
| inf isEven (2:ones)| \rightarrow* | False        |
| append ones [2,3]  | \rightarrow* | y {y \rightarrow 1:y} |

Regular Curry also supports infinite lists up to a certain extent, by means of lazy evaluation. We can, for instance, consider the infinite list ones, and reason with such infinite lists. However, reasoning on infinite lists in regular Curry is restricted to reasoning over (unbounded) finite sublists. For instance, for a conjunction over an infinite list of Boolean values, only cases in which the resulting value is False will terminate. In our approach, reasoning over (regular) infinite lists is not restricted in such a fashion. Of course, the reasoning used in our approach could be implemented in regular Curry as well, but this would require additional data types and reasoning methods to be made explicit. This would prevent a natural way of modelling.
4.2 Büchi automata

In the program $P_2$ in Figure 3 we consider Büchi automata, and in particular the example automaton given below. Note that the definitions for \texttt{inf} and \texttt{loop} are exactly the same as in $P_1$ in Figure 2. This example shows that we can directly encode the definition of a Büchi automaton, and the acceptance conditions of Büchi automata, resulting in a mechanism to decide acceptance (of regular infinite lists) of the particular automaton.

![Diagram of Büchi automaton]

Reduction rules:

\texttt{-- particular automaton}
\begin{align*}
\text{trans 1 a} &= 1 \\
\text{trans 1 a} &= 2 \\
\text{trans 2 b} &= 2 \\
\text{initial} &= 1 \\
\text{final} &= 2 \\
\end{align*}

\texttt{-- mechanism}
\begin{align*}
\text{path q s} &= q : \text{path (trans q s) ss} \\
\text{accept string} &= \text{inf } (\forall x. x \rightarrow x == \text{final}) (\text{path initial string}) \\
\text{inf} f xs &= \text{inf'} f \text{ False } (\text{loop xs}) \\
\text{inf'} f b (x:xs) &= \text{inf'} f (f x \mid b) xs \\
\text{loop } (x:xs) &= \text{loop } xs \\
\end{align*}

Assumption possibilities:
\begin{align*}
\text{inf'} _ _ b _ & \rightarrow \{ b \} \\
\text{loop } x & \rightarrow \{ x \} \\
\end{align*}

Figure 3: Example program $P_2$.

We can make the following derivations by using program $P_2$. These example derivations show that the direct encoding of Büchi automata in $P_2$ lead to a direct implementation that decides acceptance of infinite sequences. Note that this example shows that we can directly use nondeterminism (the Büchi automaton encoded in $P_2$ is nondeterministic).

| term             | value | substitution |
|------------------|-------|--------------|
| \texttt{accept} (a:a:y \{ y \rightarrow b:y \}) | \rightarrow^* True | \{x \rightarrow a:y \{ y \rightarrow b:y\}\} |
| \texttt{accept} x | \rightarrow^* True | \{x \rightarrow a:y \{ y \rightarrow b:y\}\} |
Of course, it is also possible to encode Büchi automata in regular Curry. However, encoding the acceptance conditions is not as straightforward as in our approach. In that case, namely, one has to explicitly represent (cyclic) paths and the existence of a final state in cycles of these paths.

4.3 Bisimulation in Kripke structures

In the program \( P_3 \) in Figure 4, we consider Kripke structures, and in particular the example structure given below. Note that the definitions for `and`, `forall` and `exists` are exactly the same as in \( P_1 \) in Figure 2. This example shows that we can directly encode the definition of several Kripke structures, and the bisimulation conditions of Kripke structures, resulting in a mechanism to decide bisimulation of the particular Kripke structures.

We can make the following derivations by using program \( P_3 \). These example derivations illustrate that the direct encoding of the bisimulation conditions directly lead to a decision procedure.

| term | value | substitution |
|------|------|-------------|
| bisim "m1" 1 "m2" 2 \( \rightarrow \) \* | True | \{x \rightarrow 3\} |
| bisim "m1" 1 "m2" x \( \rightarrow \) \* | True | \{x \rightarrow 3\} |

Similarly to the example of Büchi automata given above, it is possible to encode bisimulation of Kripke structures in regular Curry, but not nearly as directly and straightforwardly as in our approach.

4.4 Manipulation of infinite lists

In the program \( P_4 \) in Figure 5, we consider manipulation of infinite lists. This example shows that we can directly use the definition of several list manipulation operators from the finite case also in the infinite case (with regular terms), with the intended interpretation.

We can make the following derivations by using program \( P_4 \). These examples show that the operations `zip`, `even` and `odd` can be naturally used for infinite lists as well as finite lists. An equally natural encoding of such operations on infinite lists in regular Curry is not possible.

| term | value | substitution |
|------|------|-------------|
| zip ones twos \( \rightarrow \) \* | 1:2:y \{y \rightarrow 1:2:y\} | \{y \rightarrow 1:2:y\} |
| zip ones ones \( \rightarrow \) \* | 1:1:y \{y \rightarrow 1:1:y\} | (see below) |
| odd (zip ones x) \( \rightarrow \) \* | 1:y \{y \rightarrow 1:y\} | (see below) |
| even (zip x twos) \( \rightarrow \) \* | 2:y \{y \rightarrow 2:y\} | (see below) |
| odd ones \( \rightarrow \) \* | 1:y \{y \rightarrow 1:y\} | |
| even ones \( \rightarrow \) \* | 1:y \{y \rightarrow 1:y\} | |

No derivation for `odd (zip ones x)` and `even (zip x twos)` leading to the mentioned values has an empty substitution. Each derivation results in a substitution mapping \( x \) to an infinite
Furthermore, for every substitution mapping \( x \) to any cyclic term representing an infinite list (containing only fresh variables), there is a derivation leading to the mentioned values with this substitution.

Note that the term \( \text{nat} \ n \), for any integer \( n \), is infinite and non-regular. Our computational strategy is not suited to reason about such terms. This example illustrates this. We have that evaluating the following terms results in a diverging derivation. Similar effects occur when such non-regular infinite terms are used in the previous examples.

\[
\text{term} \\
\text{odd} (\text{zip ones} (\text{nat} \ 1)) \to^* \ldots \\
\text{even} (\text{zip} (\text{nat} \ 1) \text{twos}) \to^* \ldots
\]

5 Declarative semantics

Naturally, when modifying the operational semantics to interpret programs coinductively, we would like to change the denotational, or declarative, semantics accordingly. In this section, we suggest a possibility for suitable denotational semantics. Also, by means of several examples we illustrate how this suggested semantics differs from the inductive case. Furthermore, these examples serve to illustrate the suitability of the suggested semantics for the coinductive case.

For inductively interpreted functional logic programs, initial algebra denotational semantics is well-suited. Dually, for coinductively interpreted functional logic programs, we suggest a final coalgebra denotational semantics. For a general background on algebras and coalgebras, see for instance [9]. Consider the following (partial) signature specifying a particular type \( \text{List} \):

\[
\text{List} \to \emptyset | \mathbb{N} : \text{List} ;
\]

Intuitively, in the inductive case, terms (of type \( \text{List} \)) correspond to finite lists of natural numbers. In the coinductive case, intuitively, cyclic terms (of type \( \text{List} \)) correspond to finite or infinite lists of natural numbers.

A suitable denotational semantics for terms of type \( \text{List} \) is the initial \( F \)-algebra on the category of sets \( \text{Set} \), where the corresponding functor \( F : \{ \bot \} \cup (\mathbb{N} \times X) \to X \) is derived directly from the signature. Call this initial algebra \( IA \). Terms correspond to elements of \( IA \). In fact, the set of finite lists on \( \mathbb{N} \) is a suitable initial algebra. Now, functions from terms to natural numbers get the denotation of an \( F \)-algebra on \( \mathbb{N} \), functions from terms to terms get the denotation of an \( F \)-algebra on \( IA \). For instance, the (Curry encoding of the) function \( \text{length} : \text{List} \to \mathbb{N} \) returning the length of a list would denote the algebra \((\mathbb{N}, \alpha)\), given by:

\[
\alpha : \{ \bot \} \cup (\mathbb{N} \times \mathbb{N}) \to \mathbb{N} \\
\bot \mapsto 0 \quad (x, n) \mapsto 1 + n
\]

By initiality of \( IA \), there is exactly one morphism from \( IA \) to this algebra on \( \mathbb{N} \), which coincides with the function returning the length of lists in \( IA \). Also, for instance, pairs of terms of type \( \text{List} \) are assigned the denotation of elements in the algebra \((IA \times IA)\). In an analogous fashion, denotational semantics in algebraic terms can be assigned to the complete program.

The suggested final coalgebra semantics for cyclic terms of type \( \text{List} \) are completely dual to the initial algebra semantics for the inductive case. In this semantics, cyclic terms denote elements of the final \( G \)-coalgebra on \( \text{Set} \), where the corresponding functor \( G : X \to \{ \bot \} \cup (\mathbb{N} \times X) \) is derived directly from the signature. Call this final coalgebra \( FC \). Cyclic terms correspond to

---

1There are more such initial \( F \)-algebras on \( \text{Set} \), but they are all isomorphic.
2There are more such final \( G \)-coalgebras on \( \text{Set} \), but they are all isomorphic.
elements of $FC$. In fact, the set of all finite and infinite lists on $\mathbb{N}$ is a suitable final coalgebra. Assigning denotations to functions works dually to the inductive case. Functions from natural numbers to cyclic terms are $G$-coalgebras on $\mathbb{N}$, and functions from cyclic terms to cyclic terms are $G$-coalgebras on $FC$. For instance, the function $\text{repeat}$ from natural numbers to cyclic terms given by $\text{repeat } n = (n::(\text{repeat } n))$ would denote the coalgebra $(\mathbb{N}, \beta)$, given by:

$$\beta : \mathbb{N} \rightarrow \{\bot\} \cup (\mathbb{N} \times \mathbb{N})$$

$$n \mapsto (n, n)$$

By finality of $FC$, there is exactly one morphism from this coalgebra on $\mathbb{N}$ to $FC$, which coincides with the function mapping any natural number to the infinite list containing only this number. Also, for instance, pairs of cyclic terms are assigned the denotation of elements in the coalgebra $(FC \times FC)$. Again, analogously, the whole program can be assigned denotational semantics in coalgebraic terms.

The above exposition is a gross oversimplification, of course. Things get more intricate, for instance, when a program is interpreted partially inductively and partially coinductively. Further research is needed on the topic of denotational semantics. We merely suggest a direction for research in this area.

6 Conclusions

We showed how functional logic programming can be adapted to interpret programs coinductively as well as inductively. We singled out a particular class of (possibly infinite) objects interesting for this kind of reasoning, namely regular terms, and showed how the usual data structures can be modified to capture these objects. We showed how the operational semantics of the functional logic programming language Curry can be altered to allow for coinductive reasoning, and suggested how a suitable declarative semantics can be obtained. Furthermore, we illustrated the working and usefulness of our methods with several examples.

Working out a declarative semantics in full detail, and relating this semantics to the operational semantics, would be a topic of further research. Another direction for further research would be to investigate whether and how the computational mechanisms used in this paper could be optimized.

References

[1] M. Hanus, “A unified computation model for functional and logic programming,” in POPL, pp. 80–93, 1997.
[2] M. Hanus, “Multi-paradigm declarative languages,” in ICLP (V. Dahl and I. Niemelä, eds.), vol. 4670 of Lecture Notes in Computer Science, pp. 45–75, Springer, 2007.
[3] L. Simon, A. Bansal, A. Mallya, and G. Gupta, “Co-logic programming: Extending logic programming with coinduction,” in ICALP (L. Arge, C. Cachin, T. Jurjanzski, and A. Tarlecki, eds.), vol. 4596 of Lecture Notes in Computer Science, pp. 472–483, Springer, 2007.
[4] G. Rosu and J. Goguen, “Circular coinduction,” International Joint Conference on Automated Reasoning, 2000.
[5] D. Lucanu, E.-I. Goriac, G. Caltais, and G. Roșu, “CIRC: a behavioral verification tool based on circular coinduction,” in CALCO’09: Proceedings of the 3rd international conference on Algebra and coalgebra in computer science, Springer-Verlag, Sept. 2009.

[6] A. Martelli and U. Montanari, “An efficient unification algorithm,” Transactions on Programming Languages and Systems, vol. 4, no. 2, pp. 258–282, 1982.

[7] S. Antoy, R. Echahed, and M. Hanus, “A needed narrowing strategy,” Journal of the ACM (JACM), vol. 47, no. 4, pp. 776–822, 2000.

[8] L. E. Simon, Extending logic programming with coinduction. PhD thesis, University of Texas at Dallas, Richardson, TX, USA, 2006. AAI3224400.

[9] B. Jacobs and J. Rutten, “A tutorial on (co)algebras and (co)induction,” EATCS Bulletin, vol. 62, pp. 62–222, 1997.
A Example derivations

In Figure 6 we include several derivations for the example programs given in Section 4. For each derivation, we indicate certain (meaningful) steps in the derivation, together with the cases of the function \( cs \) used to get from the previous step to this step (indicated in the column titled case of \( cs \)), and the (partial) resulting substitution calculated in the derivation from the previous step to this step (indicated in the column titled \( \sigma \)).
Reduction rules:

```
-- kripke structures
state "m1" = 1
tran "m1" 1 = 1
val "m1" 1 = 'p'

state "m2" = 2
state "m2" = 3
tran "m2" 2 = 2
tran "m2" 2 = 3
tran "m2" 3 = 2
tran "m2" 3 = 3
val "m2" 2 = 'p'
val "m2" 3 = 'p'

-- mechanism
bisim m1 w1 m2 w2 =
  let next1 = findall (tran m1 w1)
  and next2 = findall (tran m2 w2) in
  sameSet (findall (val m1 w1)) (findall (val m2 w2)) &&
  forall (\v1 -> (exists (\v2 -> bisim m1 v1 m2 v2) next2) next1) &&
  forall (\v2 -> (exists (\v1 -> bisim m1 v1 m2 v2) next1) next2)

and [] = True
and (x:xs) = x && (and xs)
foral f xs = and (map f xs)
exists f xs = not (and (map (\x -> not (f x)) xs))

sameSet xs ys = (subSet xs ys) && (subSet ys xs)
subSet [] _ = True
subSet (x:xs) ys = (elem x ys) && subSet xs ys

Assumption possibilities:
and _ _ _ _ ~> {True}
bisim _ _ _ _ ~> {True}
```

Figure 4: Example program $P_3$. 

23
Reduction rules:

\[
\begin{align*}
\text{zip } [] & \text{ ys } = \text{ ys} \\
\text{zip } (x:xs) & \text{ (y:ys) } = \text{ x:y:(zip } xs \text{ ys)} \\
\end{align*}
\]

\[
\begin{align*}
\text{odd } [] & = [] \\
\text{odd } (x:xs) & = x:(\text{even } xs) \\
\end{align*}
\]

\[
\begin{align*}
\text{even } [] & = [] \\
\text{even } (x:xs) & = \text{odd } xs \\
\end{align*}
\]

\[
\begin{align*}
\text{ones} & = 1:\text{ones} \\
\text{twos} & = 2:\text{twos} \\
\text{nat } n & = n:(\text{nat } (n+1)) \\
\end{align*}
\]

Assumption possibilities:

\[
\begin{align*}
\text{zip } _ & \Rightarrow \{\_\} \\
\text{odd } _ & \Rightarrow \{\_\} \\
\end{align*}
\]

Figure 5: Example program \( P_4 \).
term case(s) of cs σ
for all isOdd ones
→  and (map isOdd ones) (3.1)
→ and ((isOdd 1):(map isOdd ones)) (3.1)
→ and (True:(map isOdd ones)) (3.1)
→ True && (and (map isOdd ones)) (3.1)
→ and (map isOdd ones) (3.1)
→* True (3.2), (2), (3.1)

term case(s) of cs σ
append ones [2,3]
→ l:(append ones [2,3]) (3.1)
→ (x1 == 1:x1) ⇒ x1 (3.2)
→ True ⇒ y1 {y1 → 1:y1} (2) (x1 → y1 {y1 → 1:y1})
→ y1 (y1 → 1:y1) (3.1)

term case(s) of cs σ
accept x
→ inf (==final) (path initial x) (3.1)
→ inf' (==final) False (loop (path initial x)) (3.1)
→ inf' (==final) False (loop (1:(path 2 x2))) (3.1) {x → (a:x2)}
→ inf' (==final) False (loop (path 2 x2)) (3.1)
→ inf' (==final) False (loop (2:(path 2 x3))) (3.1) {x2 → (b:x3)}
→ inf' (==final) False (loop (path 2 x3)) (3.1)
→* inf' (==final) False (path 2 y1) {y1 → b:y1} (3.2), (2), (3.1) (x3 → y1 {y1 → b:y1})
→ inf' (==final) ((2==final) || False) (path 2 y1) (3.1)
→* inf' (==final) True (path 2 y1) {y1 → b:y1} (3.1)
→* inf' (==final) True (path 2 y1) {y1 → b:y1} (3.1)
→* True (3.2), (2), (3.1)

term case(s) of cs σ
odd (zip ones x)
→ odd (1:(odd (zip ones xs1))) (3.1) {x → (x1:xs1)}
→* 1:(odd (zip ones xs1)) (3.1)
→ (x2 == 1:x2) ⇒ x2 (3.2) {x → y1, xs1 → y1}
→ True ⇒ y2 {y2 → 1:y2} (2) {y1 → x1:y1})
→ y2 (y2 → 1:y2) (3.1)

Figure 6: Example derivations.