Corecursive Featherweight Java Revisited

Pietro Barbieri[1], Francesco Dagnino[1], Elena Zucca[1], and Davide Ancona[1]

DIBRIS, University of Genova

Abstract. We describe a Java-like calculus which supports cyclic data structures, and offers a mechanism of *flexible regular corecursion* for their manipulation. The calculus enhances an earlier proposal by a more sophisticated reduction semantics, which filters out, by an additional check, some spurious results which were obtained in the previous model.

Keywords: operational semantics · coinduction · programming paradigms.

1 Introduction

Recently, several approaches to support non-well-founded data structures and their manipulation have been proposed in all main programming paradigms: logic [12,1,4], functional [11] and object-oriented [5,6].

Notably, coFJ [5,6] is an extension of Featherweight Java (FJ) [9], the reference calculus for Java-like languages, where it is possible to define cyclic objects, and methods are equipped with a mechanism of *flexible regular corecursion* for manipulating such objects.

However, the operational semantics provided in [5,6] failed to model the intuitively expected behaviour in some problematic cases. Here, we provide a more sophisticated reduction semantics, where they are correctly handled. More precisely, the revised semantics filters out spurious results by an additional check.

This paper presents ongoing work. In a forthcoming full version, we plan to formally state and prove the relation of the revised semantics with a more abstract semantics provided by generalized inference systems [3,8], as detailed in Section 6.

In this section we briefly recall coFJ by some simple examples, and informally illustrate its semantics. In Section 2 we report the (original) formal definition of coFJ, and in Section 3 we discuss problematic examples. In Section 4 we provide the revised semantics. Finally, in Section 6 we outline related and further work.

The coFJ calculus is purely functional like FJ, but, differently from FJ, it is possible to manipulate cyclic objects, since values can take an equational shape; for instance, X=**`new`** $C(X)$ is an instance of class C whose unique field contains the object itself. Furthermore, methods are *regularly corecursive*. This means that execution keeps trace of the pending method calls, so that, when a call, say, $v.m(v_1, \ldots, v_n)$, is encountered the second time, this is detected, avoiding non-termination as it would happen with ordinary recursion. This mechanism mimics what happens in *co-SLD resolution* [12,13,4], where already encountered goals, called (dynamic) coinductive hypotheses, are considered successful. However, in coFJ regular corecursion is *flexible* since the behaviour of the method when a cycle is detected can be specified by the programmer. To this end, a method body is a pair of two expressions: the standard (inductive) definition, and an additional *codefinition*.

Consider as a first example the following class declarations:[1]

```
class List extends Object { }
class EmptyList extends List { }
class NonEmptyList extends List { int head; List tail; }
class ListFactory extends Object {
    NonEmptyList repeat(int n) {
        new NonEmptyList(n,this.repeat(n)) }
    NonEmptyList zeroOne() {
        new NonEmptyList(0,this.oneZero()) }
    NonEmptyList oneZero() {
        new NonEmptyList(1,this.zeroOne()) }
}
```

Finite lists are constructed as in FJ, e.g., `new NonEmptyList(2,new EmptyList())`. However, in coFJ it is also possible to construct cyclic lists, by invoking *lf*`.repeat(0)` with *lf*`=new ListFactory()`; such a call would not terminate with the standard FJ (and Java) semantics, whereas it returns a well-defined value in coFJ, that is, the cyclic object `L=new NonEmptyList(0,L)`, thanks to regular corecursion. Indeed, the operational semantics presented in Section 2 keeps trace of pending calls, each one uniquely identified by a fresh *label*. In this way, if the label, say, `L`, is generated for the initial call *lf*`.repeat(0)`, when the same call is encountered the second time, `L` is returned as result, hence the result of the original call is `L=new NonEmptyList(0,L)`.

Similarly, `zeroOne()`, and `oneZero()` return the cyclic lists
$\mathtt{L}_{01}$ `= new NonEmptyList(0,new NonEmptyList(1,`$\mathtt{L}_{01}$`))` and
$\mathtt{L}_{10}$ `= new NonEmptyList(1,new NonEmptyList(0,`$\mathtt{L}_{10}$`))`, respectively.

Consider now the method `allPos` which returns true iff all the elements of the list are positive. The following definition works correctly for both non cyclic and cyclic lists.

```
class EmptyList extends List {
    boolean allPos() { true }
}
class NonEmptyList extends List { ...
    boolean allPos() {
      if(this.el <= 0) false else this.tail.allPos() }
    corec { true }  // codefinition
```

If the list is finite, then no regular corecursion is involved, since the same call cannot occur more than once; the same if the list is cyclic, but contains a non positive element, hence the method invocation returns `false`. The only case requiring regular corecursion is when the method is invoked on a cyclic list with all positive elements. In this

[1] In the examples we will use additional standard language features not considered in the formalization.

case, the approach based on the call trace described above would lead to the result `R=R`, that is, undetermined (in other words, both **`true`** and **`false`** are solutions of the equation corresponding to the method call). Instead, the codefinition specifies that **`true`** should be chosen.

The pattern used for defining method `member` is similar, but in this case the **`corec`** clause returns **`false`**.

```
class EmptyList extends List {
    boolean member(int i) { false }
}
class NonEmptyList extends List { ...
    boolean member(int i) {
      if(this.el == i) true; else this.tail.member(i) }
    corec { false }
}
```

We show now an example of codefinition returning an object rather than a primitive value: the method `noRep` which, invoked on a list, returns the finite (non cyclic) list with no repeated elements.

```
class EmptyList extends List {
    EmptyList noRep() { new EmptyList() }
}
class NonEmptyList extends List { ...
    List noRep() {
        let l = this.tail.noRep() in
            if (l.member(this.el)) l
            else new NonEmptyList(this.el,l) }
    corec { new EmptyList() }
}
```

For brevity we have used the **`let in`** construct, with the standard obvious semantics. Note that, in case `noRep` is invoked on the cyclic list `L = new NonEmptyList(0,L)`, if there was no **`corec`** clause, the result of `this.tail.noRep()` would be undetermined, hence `l.member(this.el)` could not be computed.

2 Calculus

In this section we report coFJ syntax (Figure 1) and operational semantics (Figure 2) from [5,6]. Notations and conventions follow those of FJ. We assume infinite sets of *class names* C, including the special class name `Object`, *field names* f, *method names* m, *variables* x, including the special variables `this` and `any`, and *labels* X. We write $\overline{cd}$ as a shorthand for a possibly empty sequence $cd_1 \ldots cd_n$, and analogously for other sequences. For simplicity, the codefinition e' is statically restricted to avoid recursive (even indirect) calls to the same method, and can use the special variable `any`. A method declaration $C\ m(\overline{C\ x})\ \{e\}$ is an abbreviation for $C\ m(\overline{C\ x})\ \{e\}$ **`corec`** $\{$`any`$\}$.

Values and objects (class instances) coincide in FJ, and have shape **`new`** $C(\overline{v})$. Here, values are generalized to the form X_1= ... X_n=**`new`** $C(\overline{v})$, abbreviated $\overline{X}$ = **`new`** $C(\overline{v})$

p	$::=$	$\overline{cd}\ e$	program
cd	$::=$	$\texttt{class}\ C\ \texttt{extends}\ C'\ \{\ \overline{fd}\ \overline{md}\ \}$	class declaration
fd	$::=$	$C\ f\texttt{;}$	field declaration
md	$::=$	$C\ m(\overline{C\ x})\ \{e\}\ \texttt{corec}\ \{e'\}$	method declaration
e	$::=$	$x \mid e.f \mid \texttt{new}\ C(\overline{e}) \mid e.m(\overline{e})$	expression
v	$::=$	$\texttt{new}\ C(\overline{v}) \mid X{=}v \mid X$	value (object)
c	$::=$	$v.m(\overline{v})$	call

Fig. 1. coFJ syntax

with our convention, so that cyclic objects can be represented. Values which, intuitively, represent the same cyclic object, as, for instance, the following:

```
new C(Y=X=new C(new C(X)))
Y=new C(X=new C(Y))
Z=new C(Z)
```

are considered equal. Moreover, values of shape $X_1{=}\ldots\ X_n{=}X_i$, with $i \in 1..n$, are all representations of "undetermined".

Note that values representing cyclic objects are *not* expressions, since, to keep the language minimal, they can only be obtained as result of a method invocation, as shown in the examples of previous section. Values which are not objects (notably, representations of "undetermined" and labels) cannot be safely used as receivers in field accesses and method invocations.

In the following, a *call* c is a method invocation where receiver and arguments have been evaluated, that is, of shape $v.m(\overline{v})$.

The big-step semantics $e, \tau \Downarrow v$ returns the result v, if any, of evaluating an expression e in the context of a *call trace* τ keeping track of pending calls. Formally, τ is a map from calls to labels X. We prefer a big-step style since small-step semantics would require to explicitly handle stacks of call traces.

The reduction rules are given in Figure 2. We omit technical details which are as in FJ, notably, the formal definitions of parallel substitution and auxiliary functions *fields* and *mbody*. The function *co-mbody* is the analogous of *mbody* for codefinitions.

We use the following notations for maps (e.g., call traces): we write maps as sequences of pairs of shape $v.m(\overline{v}){:}X$, and $\tau\{\tau'\}$ denotes the map which is τ' where it is defined, τ elsewhere. Finally, we sometimes use the wildcard _ when the corresponding meta-variable is not relevant.

Rule (FIELD) models field access. Recall that, with the FJ convention, $\overline{f}$ stands for $f_1 \ldots f_n$. The result of the receiver expression is expected to be an object. If the selected field is actually a field of its class, then the corresponding value is returned as result. Note that this value could contain references to the enclosing receiver object, which must be unfolded. For instance, given `class C extends Object { C f; }`, if v = `X=new C(Y=new C(X))`, then v`.f` is reduced to `(Y=new C(X))`$[v/$`X`$]$ $\equiv$ `Y=new C(X=new C(Y=new C(X)))`.

$$
\text{(VAL)}\,\frac{}{v, \tau \Downarrow v}
\qquad
\text{(FIELD)}\,\frac{e, \tau \Downarrow v}{e.f, \tau \Downarrow v_i[v/\overline{X}]}
\begin{array}{l}
v = \overline{X =}\,\mathbf{new}\ C(\overline{v}) \\
\mathit{fields}(C) = \overline{f} \\
f = f_i, i \in 1..n
\end{array}
$$

$$
\text{(NEW)}\,\frac{\overline{e}, \tau \Downarrow \overline{v}}{\mathbf{new}\ C(\overline{e}), \tau \Downarrow \mathbf{new}\ C(\overline{v})}\ \mathit{fields}(C) = \overline{f}
$$

$$
\text{(INVK)}\,\frac{e_0, \tau \Downarrow v_0 \quad \overline{e}, \tau \Downarrow \overline{v} \quad e[v_0/\mathtt{this}][\overline{v}/\overline{x}], \tau\{c{:}X\} \Downarrow v}{e_0.m(\overline{e}), \tau \Downarrow X{=}v}
\begin{array}{l}
v_0 = \overline{X =}\,\mathbf{new}\ C(\_) \\
\mathit{mbody}(C, m) = (\overline{x}, e) \\
c \equiv v.m(\overline{v}) \notin \mathit{dom}(\tau) \\
X \notin \mathit{img}(\tau)
\end{array}
$$

$$
\text{(COREC)}\,\frac{e_0, \tau \Downarrow v_0 \quad \overline{e}, \tau \Downarrow \overline{v} \quad e'[v_0/\mathtt{this}][\overline{v}/\overline{x}][X/\mathtt{any}], \tau \Downarrow v_{co}}{e_0.m(\overline{e}), \tau \Downarrow v_{co}}
\begin{array}{l}
v_0 = \overline{X =}\,\mathbf{new}\ C(\_) \\
\mathit{co\text{-}mbody}(C, m) = (\overline{x}, e') \\
\tau(v.m(\overline{v})) = X
\end{array}
$$

Fig. 2. Original COFJ semantics

Rule (NEW) is the standard rule for constructor invocation. Note that with the FJ convention $\overline{f}$ stands for $f_1 \ldots f_n$ and $\overline{v}$ stands for $v_1 \ldots v_n$, hence the side condition ensures that the constructor is invoked with the appropriate number of arguments.

There are two rules for method invocation. In both, the receiver and argument expressions are evaluated first to obtain the call $c \equiv v.m(\overline{v})$, the result v of the receiver expression is expected to be an object, and method look-up is performed, starting from its class, by the standard function mbody. Then, the behavior is different depending whether such call has been already encountered.

If this is not the case, then the method invocation is handled as usual by rule (INVK): the result is obtained by evaluating the left expression e in the body (definition) where the receiver object replaces `this` and the arguments replace the parameters. Evaluation of e is performed in the call trace τ updated with the call c, associated with a fresh label X. Finally, when the evaluation of e is completed, references to the label X in the resulting value, possibly occurring in case of termination by regular corecursion of the method invocation, see rule (COREC), are bound. In this way a cyclic object can be obtained as result.

Rule (COREC) is applied when an already encountered call is detected; in this case, the right expression e' in the body (codefinition) is evaluated where the receiver object replaces `this`, the arguments replace the parameters, and, moreover, the label X found in the call trace replaces `any`.

We conclude this section by showing an example of reduction. Consider the method `allPos()` defined in Section 1. Set 1^ω the cyclic list `L = new NonEmptyList(1,L)`, and `a` the method call 1^ω`.allPos()`. The proof tree for the judgment `a`$, \emptyset \Downarrow$ `true` is shown in Figure 3. To save space, in proof trees we use abbreviated names.

$$\text{(INVK)}\dfrac{\ldots \quad \text{(IF-FALSE)}\dfrac{\text{(<)}\dfrac{\ldots}{1^\omega.\texttt{hd} < 0, \texttt{a}{:}X \Downarrow \texttt{false}} \quad \text{(COREC)}\dfrac{\text{(FIELD)}\dfrac{\text{(VAL)}\dfrac{}{1^\omega, \texttt{a}{:}X \Downarrow 1^\omega}}{1^\omega.\texttt{tl}, \texttt{a}{:}X \Downarrow 1^\omega} \quad \text{(VAL)}\dfrac{}{\texttt{true}, \texttt{a}{:}X \Downarrow \texttt{true}}}{1^\omega.\texttt{tl.allPos}(), \texttt{a}{:}X \Downarrow \texttt{true}}}{\texttt{if} \ \ldots, \texttt{a}{:}X \Downarrow \texttt{true}}}{\texttt{a}, \emptyset \Downarrow \texttt{true}}$$

Fig. 3. Reduction of 1^ω.`allPos`()

3 Problematic examples

In this section we show some examples that, according to the semantics presented in Section 2, can lead to spurious results.

In the first example, our aim is to write a method that, given a list of integers, returns the sum of all its elements. As first attempt, one could write the following coFJ definition.

```
class EmptyList extends List {
    int sum() { 0 }
}
class NonEmptyList extends List { ...
     int sum() { this.head + this.tail.sum() }
     corec { 0 }
}
```

This definition works well for the lists where the sum can actually be computed, that is, finite lists or cyclic lists where the cycle has sum 0. However, for cyclic lists where the cycle has sum different from 0, hence the result should be undefined, a spurious result is returned.

For instance, set 1^ω the cyclic list `L = new NonEmptyList(1,L)`, and s the method call 1^ω.`sum`(), it is possible to build a proof tree for the judgment $s, \emptyset \Downarrow 1$. The proof tree is shown in Figure 4.

$$\text{(INVK)}\dfrac{\text{(VAL)}\dfrac{}{1^\omega, \emptyset \Downarrow 1^\omega} \quad \text{(SUM)}\dfrac{\text{(FIELD)}\dfrac{\text{(VAL)}\dfrac{}{1^\omega, s{:}X \Downarrow 1^\omega}}{1^\omega.\texttt{hd}, s{:}X \Downarrow 1} \quad \text{(COREC)}\dfrac{\text{(FIELD)}\dfrac{\text{(VAL)}\dfrac{}{1^\omega, s{:}X \Downarrow 1^\omega}}{1^\omega.\texttt{tl}, s{:}X \Downarrow 1^\omega} \quad \text{(VAL)}\dfrac{}{0, s{:}X \Downarrow 0}}{1^\omega.\texttt{tl.sum}(), s{:}X \Downarrow 0}}{1^\omega.\texttt{hd} + 1^\omega.\texttt{tl.sum}(), s{:}X \Downarrow 1}}{s, \emptyset \Downarrow 1}$$

Fig. 4. Reduction of 1^ω.`sum`()

Note that in any case 1 *cannot be accepted as result* since it is not a solution of the equation $R = 1 + R$ generated by the method call 1^ω.`sum`().

As a second problematic example, we consider the method `remNeg()` that removes negative elements. A possible coFJ definition could be:

```
class EmptyList extends List {
    EmptyList remNeg() { new EmptyList() }
}
class NonEmptyList extends List { ...
    List remNeg() {
        if(this.head < 0) this.tail.remNeg()
        else new NonEmptyList(this.head,this.tail.remNeg())}
    corec { new EmptyList() }
```

Also this definition fails to achieve the desired behaviour. Indeed, set r the method call $1^\omega.\texttt{remNeg}()$, and $[1]$ the list `new NonEmptyList(1,new EmptyList())`, it is possible to build a proof tree for the judgment $r, \emptyset \Downarrow [1]$, which is intuitively wrong since $1^\omega.\texttt{remNeg}()$ should be reduced, instead, to 1^ω. The proof tree is shown in Figure 5. Again, the result $[1]$ is not a solution of the equation $R = \texttt{new NonEmptyList(1, } R\texttt{)}$ generated by the method call $1^\omega.\texttt{remNeg}()$.

$$\text{(INVK)}\dfrac{\ldots\quad \text{(IF-FALSE)}\dfrac{\ldots\quad \text{(NEW)}\dfrac{\ldots\quad\text{(COREC)}\dfrac{\text{(FIELD)}\dfrac{\text{(VAL)}\dfrac{}{1^\omega, r{:}X \Downarrow 1^\omega}}{1^\omega.\texttt{tl}, r{:}X\Downarrow 1^\omega}\quad \text{(NEW)}\dfrac{}{\texttt{new EList}(), r{:}X\Downarrow \texttt{new EList}()}}{1^\omega.\texttt{tl.rN}(), r{:}X\Downarrow\texttt{new EList}()}}{\texttt{new NEList}(1^\omega.\texttt{hd}, 1^\omega.\texttt{tl.rN}()), r{:}X\Downarrow[1]}}{\texttt{if} \ldots, r{:}X\Downarrow[1]}}{r,\emptyset\Downarrow[1]}$$

Fig. 5. Reduction of $1^\omega.\texttt{remNeg}()$

Note that in the first example the equation generated by the call has no solution, whereas in the second it has exactly one solution which, however, is different from the result obtained using the codefinition. Both examples suggest that the semantics presented in Section 2 should be refined by a check that the result v obtained for a method call using the codefinition is actually a solution of the corresponding equation. That is, assuming v as result of the (recursive) call, we get v in turn as result. This check is performed by the revised semantics presented in Section 4.

4 Revised calculus

The basic idea of the revised semantics is that the rule for method invocation (INVK) in Figure 2 should be refined, adding a check that the obtained result $X{=}v$ is a solution of the equation corresponding to the call $c \equiv v.m(\overline{v})$. That is, the definition in the method body, evaluated assuming $X{=}v$ as result of (recursive) calls c, should in turn give $X{=}v$ as result. Note that this is necessary only when the result $X{=}v$ has been obtained by regular corecursion (that is, the same call c has been encountered and the codefinition for c has been evaluated), otherwise the result (which can be written as v in this case) has been obtained by standard recursion and no check is needed.

To this end, in the revised semantics, the judgment has shape $e, \tau, \rho \Downarrow v, S$. In addition to the call trace τ, which plays the same role as before, there are two other auxiliary components: a map ρ from calls into values, and a set of labels S. The fact that a call c has an associated value v in ρ means that the evaluation of e is performed assuming v

as result of such (recursive) call. On the other hand, a label X belongs to S if the result of corresponding call has been obtained by regular corecursion, hence the additional check described above needs to be performed. Hence, every time a codefinition is evaluated, the label X corresponding to the call is added in S, see rule (COREC). Then, in rule (INVK-CHECK), if S contains X then the check is performed.

The rules of the revised semantics are given in Figure 6.

$$\text{(VAL)}\ \frac{}{v,\tau,\rho \Downarrow v,\emptyset} \qquad \text{(FIELD)}\ \frac{e,\tau,\rho \Downarrow v, S}{e.f,\tau,\rho \Downarrow v_i[v/X], S} \ \begin{array}{l} v = \overline{X=}\,\mathbf{new}\ C(\overline{v}) \\ \mathit{fields}(C) = \overline{f} \\ f = f_i, i \in 1..n \end{array}$$

$$\text{(NEW)}\ \frac{e_i,\tau,\rho \Downarrow v_i, S_i \quad \forall i \in 1..n}{\mathbf{new}\ C(\overline{e}),\tau,\rho \Downarrow \mathbf{new}\ C(\overline{v}), \bigcup_{i\in 0..n} S_i} \ \mathit{fields}(C) = f_1 \ldots f_n$$

$$\text{(INVK-OK)}\ \frac{\begin{array}{c} e_i,\tau,\rho \Downarrow v_i, S_i \quad \forall i \in 0..n \\ e[v_0/\mathtt{this}][\overline{v}/\overline{x}], \tau\{c{:}X\}, \rho \Downarrow v, S \end{array}}{e_0.m(\overline{e}),\tau,\rho \Downarrow v, \bigcup_{i\in 0..n} S_i \cup S} \ \begin{array}{l} v_0 = \overline{X=}\,\mathbf{new}\ C(\_) \\ \mathit{mbody}(C,m) = (\overline{x}, e) \\ c \equiv v_0.m(\overline{v}) \notin \mathit{dom}(\tau)\cup\mathit{dom}(\rho) \\ X \notin \mathit{img}(\tau) \\ X \notin S \end{array}$$

$$\text{(INVK-CHECK)}\ \frac{\begin{array}{c} e_i,\tau,\rho \Downarrow v_i, S_i \quad \forall i \in 0..n \\ e[v_0/\mathtt{this}][\overline{v}/\overline{x}], \tau\{c{:}X\}, \rho \Downarrow v, S \\ e[v_0/\mathtt{this}][\overline{v}/\overline{x}], \tau, (\rho\{c{:}X{=}v\})[X{=}v/X] \Downarrow X{=}v, S' \end{array}}{e_0.m(\overline{e}),\tau,\rho \Downarrow X{=}v, (\bigcup_{i\in 0..n} S_i \cup S)\setminus\{X\}} \ \begin{array}{l} v_0 = \overline{X=}\,\mathbf{new}\ C(\_) \\ \mathit{mbody}(C,m) = (\overline{x}, e) \\ c \equiv v_0.m(\overline{v}) \notin \mathit{dom}(\tau)\cup\mathit{dom}(\rho) \\ X \notin \mathit{img}(\tau) \\ X \in S \end{array}$$

$$\text{(COREC)}\ \frac{\begin{array}{c} e_i,\tau,\rho \Downarrow v_i, S_i \quad \forall i \in 0..n \\ e'[v_0/\mathtt{this}][\overline{v}/\overline{x}][X/\mathtt{any}], \tau, \rho \Downarrow v_{co}, \emptyset \end{array}}{e_0.m(\overline{e}),\tau,\rho \Downarrow v_{co}, \bigcup_{i\in 0..n} S_i \cup \{X\}} \ \begin{array}{l} v_0 = \overline{X=}\,\mathbf{new}\ C(\_) \\ \mathit{co\text{-}mbody}(C,m) = (\overline{x}, e') \\ \tau(v_0.m(\overline{v})) = X \end{array}$$

$$\text{(LOOK-UP)}\ \frac{e_i,\tau,\rho \Downarrow v_i, S_i \quad \forall i \in 0..n}{e_0.m(\overline{e}),\tau,\rho \Downarrow v, \bigcup_{i\in 0..n} S_i} \ \begin{array}{l} v_0 = \overline{X=}\,\mathbf{new}\ C(\_) \\ \rho(v_0.m(\overline{v})) = v \end{array}$$

Fig. 6. COFJ revised semantics

Rules (VAL), (FIELD) and (NEW) are as before, apart from the fact that the resulting set of labels is the union of those obtained by evaluating subterms. Instead, there are now four rules for method invocation. In all of them, the receiver and argument expressions are evaluated, obtaining the call $c \equiv v.m(\overline{v})$.

Rules (INVK-OK) and (INVK-CHECK) are two different versions of the previous rule (INVK) handling a call c which is encountered the first time, as expressed by the side condition $c \notin \mathit{dom}(\tau)\cup\mathit{dom}(\rho)$. In both, the definition e, where the receiver object replaces `this` and the arguments replace the parameters, is evaluated. Such evaluation is performed in the call trace τ updated with the call c, associated to an unused label X, and produces a set of labels S. Then there are two cases, depending on the condition $X \in S$.

If $X \notin S$, then the evaluation of the definition for c has been performed without evaluating the codefinition. That is, no recursive call c has been encountered, hence the result of the evaluation has been obtained by standard recursion, and no additional check is needed.

If $X \in S$, instead, then the evaluation of the definition for c has required to evaluate the codefinition. In this case, $e[v_0/\mathtt{this}][\overline{v}/\overline{x}]$ is evaluated once more updating ρ with the assumption that $X{=}v$ is the result of c. The result obtained in this way must be in turn $X{=}v$. In case this does not happen, rule (INVK-CHECK) cannot be applied since the last premise does not hold. For simplicity, we assume the result of c to be undefined in this case; an additional rule could be added raising a runtime error in case the result obtained by the third premise is different from the expected one, as should be done in an implementation.

In both cases, the resulting set of labels is the union of those obtained by evaluating the subterms and the definition for the call. The only difference is that in rule (INVK-CHECK), since the call associated with X has been checked, the label X is removed from the resulting set of labels. Moreover, labels obtained during the additional check step are not relevant.

The remaining two rules handle a call which has already been encountered, that is, the corresponding label X is either in $dom(\tau)$ or in $dom(\rho)$.

In the former case, rule (COREC) works as in the previous semantics, that is, evaluates the codefinition where the receiver object replaces `this`, the arguments replace the parameters, and, moreover, the label X found in the call trace replaces `any`. In addition, the label associated with the call is added in S.

The latter case, instead, is introduced in the revised semantics to perform the additonal check step. To this end, rule (LOOK-UP) simply returns the associated value for a call which is present in the result table.

5 Examples

Let us consider again the examples discussed in Section 3. We show that, with the new semantics, there exists no proof tree for the judgment $s, \emptyset, \emptyset \Downarrow 1, \emptyset$. Recall that 1^ω is an abbreviation for the cyclic list `L = new NonEmptyList(1,L)`, and s for the call $1^\omega.\mathtt{sum}()$. The "tentative" proof tree is shown in Figure 7.

Comparing with the proof tree in Figure 4, note that:

- Rule (COREC) adds to the set S the label X (corresponding to the call s).
- Rule (INVK-CHECK) has the additional premise $1^\omega.\mathtt{hd}{+}1^\omega.\mathtt{tl}.\mathtt{sum}(), \emptyset, s{:}1 \Downarrow 1, \emptyset$, whose tentative proof tree T is separately shown.
- In T, there is no way to prove the third premise of rule (SUM), where the result of the method invocation should be 0. Indeed, since $s{:}1$ is in the result table, the only applicable rule for the method invocation is (LOOK-UP), but applying this rule we get 1 as result.

$$
{\scriptstyle(\text{INVK-CHECK})}\dfrac{\ldots\quad{\scriptstyle(\text{SUM})}\dfrac{\ldots\quad{\scriptstyle(\text{COREC})}\dfrac{\ldots\quad{\scriptstyle(\text{FIELD})}\dfrac{{\scriptstyle(\text{VAL})}\dfrac{}{1^\omega, s{:}X, \emptyset \Downarrow 1^\omega, \emptyset}}{1^\omega.\mathtt{tl}, s{:}X, \emptyset \Downarrow 1^\omega, \emptyset}\quad{\scriptstyle(\text{VAL})}\dfrac{}{0, s{:}X, \emptyset \Downarrow 0, \emptyset}}{1^\omega.\mathtt{tl}.\mathtt{sum}(), s{:}X, \emptyset \Downarrow 0, \emptyset \cup X}}{1^\omega.\mathtt{hd} + 1^\omega.\mathtt{tl}.\mathtt{sum}(), s{:}X, \emptyset \Downarrow 1, \{X\}}\qquad T}{s, \emptyset, \emptyset \Downarrow 1, \emptyset}
$$

$$
T \;=\; {\scriptstyle(\text{SUM})}\dfrac{{\scriptstyle(\text{FIELD})}\dfrac{{\scriptstyle(\text{VAL})}\dfrac{}{1^\omega, \emptyset, \emptyset \Downarrow 1^\omega, \emptyset}}{1^\omega.\mathtt{hd}, \emptyset, \emptyset \Downarrow 1, \emptyset}\quad{\scriptstyle(???)}\dfrac{{\scriptstyle(\text{FIELD})}\dfrac{{\scriptstyle(\text{VAL})}\dfrac{}{1^\omega, \emptyset, \emptyset \Downarrow 1^\omega, \emptyset}}{1^\omega.\mathtt{tl}, \emptyset, \emptyset \Downarrow 1^\omega, \emptyset}\quad \mathtt{NO\ RULE\ TO\ APPLY}}{1^\omega.\mathtt{tl}.\mathtt{sum}(), \emptyset, s{:}1 \Downarrow 0, \emptyset}}{1^\omega.\mathtt{hd} + 1^\omega.\mathtt{tl}.\mathtt{sum}(), \emptyset, s{:}1 \Downarrow 1, \emptyset}
$$

Fig. 7. Tentative reduction of 1^ω.sum()

For what concerns the `remNeg()` method, set r the call $1^\omega.\mathtt{rN}()$, $[1]$ the list **new** `NonEmptyList(1,`**new** `EmptyList())` and $[\,]$ the list **new** `EmptyList()`. In Figure 8 we show the "tentative" proof tree for the judgment $r, \emptyset, \emptyset \Downarrow [1], \emptyset$.

$$
{\scriptstyle(\text{INVK-CHECK})}\dfrac{\ldots\quad{\scriptstyle(\text{IF-FALSE})}\dfrac{\ldots\quad{\scriptstyle(\text{NEW})}\dfrac{\ldots\quad{\scriptstyle(\text{COREC})}\dfrac{\ldots\quad{\scriptstyle(\text{FIELD})}\dfrac{{\scriptstyle(\text{VAL})}\dfrac{}{1^\omega, r{:}X, \emptyset \Downarrow 1^\omega, \emptyset}}{1^\omega.\mathtt{tl}, r{:}X, \emptyset \Downarrow 1^\omega, \emptyset}\quad{\scriptstyle(\text{VAL})}\dfrac{}{[\,], r{:}X, \emptyset \Downarrow [\,], \emptyset}}{1^\omega.\mathtt{tl}.\mathtt{rN}(), r{:}X, \emptyset \Downarrow [\,], \emptyset \cup X}}{\mathtt{new\ NEList}(1^\omega.\mathtt{hd}, 1^\omega.\mathtt{tl}.\mathtt{rN}()), r{:}X, \emptyset \Downarrow [1], \{X\}}}{\mathtt{if}\ \ldots, r{:}X, \emptyset \Downarrow [1], \{X\}}\qquad T}{r, \emptyset, \emptyset \Downarrow [1], \emptyset}
$$

$$
T \;=\; {\scriptstyle(\text{IF-FALSE})}\dfrac{\ldots\quad{\scriptstyle(\text{NEW})}\dfrac{{\scriptstyle(\text{FIELD})}\dfrac{{\scriptstyle(\text{VAL})}\dfrac{}{1^\omega, \emptyset, \emptyset \Downarrow 1^\omega, \emptyset}}{1^\omega.\mathtt{hd}, \emptyset, \emptyset \Downarrow 1, \emptyset}\quad{\scriptstyle(???)}\dfrac{{\scriptstyle(\text{FIELD})}\dfrac{{\scriptstyle(\text{VAL})}\dfrac{}{1^\omega, \emptyset, \emptyset \Downarrow 1^\omega, \emptyset}}{1^\omega.\mathtt{tl}, \emptyset, \emptyset \Downarrow 1^\omega, \emptyset}\quad \mathtt{NO\ RULE\ TO\ APPLY}}{1^\omega.\mathtt{tl}.\mathtt{rN}(), \emptyset, r{:}[1] \Downarrow [\,], \emptyset}}{\mathtt{new\ NEList}(1^\omega.\mathtt{hd}, 1^\omega.\mathtt{tl}.\mathtt{rN}()), \emptyset, r{:}[1] \Downarrow [1], \emptyset}}{\mathtt{if}\ \ldots, \emptyset, r{:}[1] \Downarrow [1], \emptyset}
$$

Fig. 8. Tentative reduction of 1^ω.rN()

Again, the judgment has no proof tree because the check performed by rule (INVK-CHECK) fails.

As last example, we show a case where the check succeeds, considering `sum` as in Section 3. We set 0^ω the cyclic list `L = `**new** `NonEmptyList(0,L)`, and t the call $0^\omega.\mathtt{sum}()$. The proof tree for the judgment $t, \emptyset, \emptyset \Downarrow 0, \emptyset$ is shown in Figure 9.

Note that the `sum` method, interpreted with the revised semantics, returns the desired result: the sum of the elements if the list is either finite or terminating with a cycle of sum 0, undefined otherwise. The `remNeg` method, instead, if the list has a cycle with non negative elements, returns undefined (a runtime error in an implementation) rather than a wrong result as it was in the original semantics. That is, the programmer is alerted

$$\text{(INVK-CHECK)}\dfrac{\text{(VAL)}\dfrac{}{0^\omega,\emptyset,\emptyset \Downarrow 0^\omega,\emptyset} \qquad \text{(SUM)}\dfrac{\ldots \quad \text{(COREC)}\dfrac{\text{(FIELD)}\dfrac{\text{(VAL)}\dfrac{}{0^\omega, t{:}X,\emptyset \Downarrow 0^\omega,\emptyset}}{0^\omega.\texttt{tl}, t{:}X,\emptyset \Downarrow 0^\omega,\emptyset} \quad \text{(VAL)}\dfrac{}{0, t{:}X,\emptyset \Downarrow 0,\emptyset}}{0^\omega.\texttt{tl.sum}(), t{:}X,\emptyset \Downarrow 0,\emptyset \cup X}}{0^\omega.\texttt{hd} + 0^\omega.\texttt{tl.sum}(), t{:}X,\emptyset \Downarrow 0,\{X\}} \qquad T}{t,\emptyset,\emptyset \Downarrow 0,\emptyset}$$

$$T \;=\; \text{(SUM)}\dfrac{\text{(FIELD)}\dfrac{\text{(VAL)}\dfrac{}{0^\omega,\emptyset,\emptyset \Downarrow 0^\omega,\emptyset}}{0^\omega.\texttt{hd},\emptyset,\emptyset \Downarrow 0,\emptyset} \quad \text{(LOOK-UP)}\dfrac{\text{(FIELD)}\dfrac{\text{(VAL)}\dfrac{}{0^\omega,\emptyset,\emptyset \Downarrow 0^\omega,\emptyset}}{0^\omega.\texttt{tl},\emptyset,\emptyset \Downarrow 0^\omega,\emptyset}}{0^\omega.\texttt{tl.sum}(),\emptyset, t{:}0 \Downarrow 0,\emptyset}}{0^\omega.\texttt{hd} + 0^\omega.\texttt{tl.sum}(),\emptyset, t{:}0 \Downarrow 0,\emptyset}$$

Fig. 9. Reduction of 0^ω`.sum()`

that there is something to be corrected. A `remNeg` definition which returns the desired result also in this case is the following:

```
class EmptyList extends List {
    boolean allNeg() {true}
    EmptyList remNeg() { new EmptyList() }
}
class NonEmptyList extends List { ...
    boolean allNeg() { this.head < 0 && this.tail.allNeg() }
    corec { true }
    List remNeg() {
        if(this.head < 0) this.tail.remNeg()
        else new NonEmptyList(this.head,this.tail.remNeg())}
    corec { if (this.allNeg()) new EmptyList() else any }
}
```

The COFJ (revised) operational semantics is an extension of the FJ semantics. That is, evaluation in the COFJ operational semantics of FJ programs gives exactly the same results of FJ evaluation, as formally stated below.

Theorem 1 (Conservativity). *Given an* FJ *class table, for* e *expression in* FJ, $e \Downarrow_{\text{FJ}} v \iff e,\emptyset,\emptyset \Downarrow v,\emptyset$.

6 Conclusion

The Java-like calculus presented in this paper promotes a novel programming style, which smoothly incorporates support for cyclic data structures and coinductive reasoning in the object-oriented paradigm. The calculus enhances an earlier proposal by filtering out, by an additional runtime check, results of a (cyclic) call which are not acceptable since they are not solutions of the corresponding equation.

This paper presents ongoing work. In a forthcoming full version, we plan to formally state and prove the relation of the revised semantics with the more abstract semantics provided by *generalized inference systems* [3,8], a recently proposed formalism allowing definition of relations by mixing induction and coinduction. Besides standard rules, generalized inference systems are enriched by *corules* which play a special role.

The semantics of flexible regular corecursion presented in this paper is expected to be the operational counterpart of that obtained by considering recursive functions as relations, and recursive definitions (with codefinition) as inference systems (with corules). Of course the latter semantics is more abstract, notably relations can involve not only cyclic data structures (such as the 1^ω example) but arbitrary non-well-founded structures (such as the list of natural numbers). We plan to make this correspondence precise, proving that the operational semantics in this paper is *sound* with respect to the interpretation based on generalized inference systems, and investigating restrictions which guarantee *completeness* as well.

As already mentioned, the idea of *regular corecursion* (keeping trace of pending method calls, so to detect cyclic calls), originates from co-SLD resolution [12,13,4]. Making regular corecursion *flexible* means that the programmer can specify the behaviour in case a cycle is detected. Language constructs to achieve such flexibility have been proposed in the logic [1,2], object-oriented [5,6] and functional [11] paradigm. In the logic paradigm, the programmer can write special clauses corresponding to the corules mentioned above, so that, when a goal G is encountered the second time, standard SLD resolution of G is triggered in the program enriched by the corules. Similarly, in the COFJ calculus introduced in [5,6] and refined in this paper, the programmer can write a *codefinition* which is evaluated when a cycle is detected. In the functional paradigm, there exists a fully-fledged proposal, CoCaml (`www.cs.cornell.edu/Projects/CoCaml`), a variant of Caml supporting non-well-founded data types and corecursive methods. In CoCaml, flexibility is achieved by a different mechanism: roughly, the equation corresponding to a (recursive) call is generated, and then a solution is found through a *solver* which can be predefined or written by the programmer.

There are several other directions for further research, both on the foundational and applicative side.

First of all, we plan to investigate different approaches to the model of values, possibly relying on the *capsule* notion proposed in [10].

A challenging long-term goal is to overcome the restriction to cyclic data structures mentioned above by extending equations $X{=}v$, denoting regular terms, considered in this paper, to *algebraic equations* [7]. The approach obtained in this way should nicely combine the advantages of *lazy evaluation* with those of regular corecursion. With lazy evaluation, arbitrary non-well-founded data structures can be represented, e.g., in Haskell we can write `from n = n : from(n+1)`. However, we cannot compute results which need to explore the whole structure, whereas, with regular corecursion, this becomes possible for cyclic structures: for instance we can compute `allPos ones`, which diverges in Haskell.

On the more practical side, we plan to implement the calculus presented in this paper. This could be done directly, or through a translation in logic programming relying on a meta-interpreter supporting corules[2]. Moreover, we could possibly apply the work done on toy languages to develop an extension of a real language.

References

1. Davide Ancona. Regular corecursion in Prolog. *Computer Languages, Systems & Structures*, 39(4):142–162, 2013.

2. Davide Ancona, Francesco Dagnino, and Elena Zucca. Extending coinductive logic programming with co-facts. In Ekaterina Komendantskaya and John Power, editors, *First Workshop on Coalgebra, Horn Clause Logic Programming and Types, CoALP-Ty'16*, volume 258 of *Electronic Proceedings in Theoretical Computer Science*, pages 1–18. Open Publishing Association, 2017.
3. Davide Ancona, Francesco Dagnino, and Elena Zucca. Generalizing inference systems by coaxioms. In Hongseok Yang, editor, *26th European Symposium on Programming, ESOP 2017*, volume 10201 of *Lecture Notes in Computer Science*, pages 29–55. Springer Verlag, 2017.
4. Davide Ancona and Agostino Dovier. A theoretical perspective of coinductive logic programming. *Fundamenta Informaticae*, 140(3-4):221–246, 2015.
5. Davide Ancona and Elena Zucca. Corecursive Featherweight Java. In *FTfJP'12 - Formal Techniques for Java-like Programs*, pages 3–10. ACM Press, 2012.
6. Davide Ancona and Elena Zucca. Safe corecursion in coFJ. In *FTfJP'13 - Formal Techniques for Java-like Programs*, page 2. ACM Press, 2013.
7. B. Courcelle. Fundamental properties of infinite trees. *Theoretical Computer Science*, 25:95–169, 1983.
8. Francesco Dagnino. Coaxioms: flexible coinductive definitions by inference systems. *Logical Methods in Computer Science*, 15(1), 2019.
9. Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *ACM Symp. on Object-Oriented Programming: Systems, Languages and Applications 1999*, pages 132–146. ACM Press, 1999.
10. Jean-Baptiste Jeannin and Dexter Kozen. Computing with capsules. *Journal of Automata, Languages and Combinatorics*, 17(2-4):185–204, 2012.
11. Jean-Baptiste Jeannin, Dexter Kozen, and Alexandra Silva. Cocaml: Functional programming with regular coinductive types. *Fundamenta Informaticae*, 150:347–377, 2017.
12. L. Simon. *Extending logic programming with coinduction*. PhD thesis, University of Texas at Dallas, 2006.
13. L. Simon, A. Bansal, A. Mallya, and G. Gupta. Co-logic programming: Extending logic programming with coinduction. In *ICALP 2007*, pages 472–483, 2007.