Sound Regular Corecursion in coFJ

Davide Ancona
DIBRIS, University of Genova, Italy
davide.ancona@unige.it

Pietro Barbieri
DIBRIS, University of Genova, Italy
pietro.barbieri@edu.unige.it

Francesco Dagnino
DIBRIS, University of Genova, Italy
francesco.dagnino@dibris.unige.it

Elena Zucca
DIBRIS, University of Genova, Italy
elena.zucca@unige.it

Abstract

The aim of the paper is to provide solid foundations for a programming paradigm natively supporting the creation and manipulation of cyclic data structures. To this end, we describe coFJ, a Java-like calculus where objects can be *infinite* and methods are equipped with a *codefinition* (an alternative body). We provide an abstract semantics of the calculus based on the framework of *inference systems with corules*. In coFJ with this semantics, FJ recursive methods on finite objects can be extended to infinite objects as well, and behave as desired by the programmer, by specifying a codefinition. We also describe an operational semantics which can be directly implemented in a programming language, and prove the soundness of such semantics with respect to the abstract one.

2012 ACM Subject Classification Theory of computation → Operational semantics; Software and its engineering → Recursion; Software and its engineering → Semantics

Keywords and phrases Operational semantics, coinduction, programming paradigms, regular terms

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2020.1

Related Version A full version of the paper is available at `https://arxiv.org/abs/2005.14085`.

Funding *Davide Ancona*: Member of GNCS (Gruppo Nazionale per il Calcolo Scientifico), INdAM (Istituto Nazionale di Alta Matematica "F. Severi")

Introduction

Applications often deal with data structures which are conceptually infinite, such as streams or infinite trees. Thus, a major problem for programming languages is how to finitely represent something which is infinite, and, even harder, how to correctly manipulate such finite representations to reflect the expected behaviour on the infinite structure.

A well-established solution is *lazy evaluation*, as, e.g., in Haskell. In this approach, the conceptually infinite structure is represented as the result of a function call, which is evaluated only as much as needed. Focusing on the paradigmatic example of streams (infinite lists) of integers, we can define `two_one = 2:1:two_one`, or even represent the list of natural numbers as `from 0`, where `from n = n:from(n+1)`. In this way, functions which only need to inspect a finite portion of the structure, e.g., getting the i-th element, can be correctly implemented. On the other hand, functions which need to inspect the whole structure, e.g., `min` getting the minimal element, or `allPos` checking that all elements are positive, have an undefined result (that is, non-termination, operationally).

More recently, a different, in a sense complementary[1], approach has been considered [17, 8, 3], which focuses on *cyclic* structures (e.g., cyclic lists, trees and graphs). They can be regarded as a particular case of infinite structures: abstractly, they correspond to regular terms (or trees), that is, finitely branching trees whose depth can be infinite, but contain only a finite set of subtrees. For instance, the list `two_one` is regular, whereas the list of natural numbers is not. Typically, cyclic data structures are handled by programming languages by relying on imperative features or ad hoc data structures for bookkeeping. For instance, we can build a cyclic object by assigning to a field of an object a reference to the object itself, or we can visit a graph by marking already encountered nodes. In this approach [17, 8, 3], instead, the programming language natively supports regular structures, as outlined below:

- Data constructors are enriched by allowing equations, e.g., $x = 2 : 1 : x$.
- Functions are *regularly corecursive*, that is, execution keeps track of pending function calls, so that, when the same call is encountered the second time, this is detected, avoiding non-termination as with ordinary recursion. For instance, when calling `min` on the list $x = 2 : 1 : x$, after an intermediate call on the list $y = 1 : 2 : y$, the same call is encountered.

Regular corecursion originates from *co-SLD resolution* [20, 21, 7], where already encountered goals (up to unification), called *coinductive hypotheses*, are considered successful. However, co-SLD resolution is not flexible enough to to correctly express certain predicates on regular terms; for instance, in the `min` example, the intuitively correct corecursive definition is not sound, because the predicate succeeds for all lower bounds of l, as shown in the following.

When moving from goals to functions calls, the same problem manifests more urgently because a result should always be provided for already encountered calls. To solve this issue, the mechanism of *flexible regular corecursion* can be adopted to allow the programmer to correctly specify the behaviour of recursive functions on cyclic structures. For instance, for function `min`, the programmer specifies that the head of the list should be returned when detecting a cyclic call; in this way, on the list $x = 2 : 1 : x$, the result of the cyclic call is 2, so that the result of the original call is 1, as expected.

Flexible regular corecursion as outlined above has been proposed in the object-oriented [8], functional [17], and logic [3] paradigms (see Section 7 for more details). However, none of these proposals provides formal arguments for the correctness of the given operational semantics, by proving that it is sound with respect to some model of the behaviour of functions (or predicates) on infinite structures. The aim of this paper is to bridge this gap, by providing solid foundations for a programming paradigm natively supporting cyclic data structures. This is achieved thanks to the recently introduced framework of *inference systems with corules* [4, 13], allowing definitions which are neither inductive, nor purely coinductive. We present the approach in the context of Java-like languages, namely on an extension of Featherweight Java (FJ) [15] called coFJ, outlined as follows:

- FJ objects are smoothly generalized from finite to infinite by interpreting their definition coinductively, and methods are equipped with a *codefinition* (an alternative body).
- We provide an abstract big-step semantics for coFJ by an inference system with corules. In coFJ with this semantics, FJ recursive methods on finite objects can be extended to infinite objects as well, and behave as desired by the programmer, by specifying a codefinition. For instance, if the codefinitions for `min` and `allPos` are specified to return the head, and `true`, respectively, then `min` returns 1 on $x = 2 : 1 : x$, and 0 on the list of the natural numbers, whereas `allPos` returns `true` on both lists.

[1] As we will discuss further in the Conclusion.

- Then, we provide an operational (hence, executable) semantics where infinite objects are restricted to regular ones and methods are regularly corecursive, and we show that such operational semantics is sound with respect to the abstract one.

At `https://person.dibris.unige.it/zucca-elena/coFJ_implementation.zip` we provide a prototype implementation of coFJ, briefly described in the Conclusion. A preliminary version of the operational semantics, with no soundness proof with respect to a formal model, has been given in [10]. An extended version of the paper including proofs can be found at `https://arxiv.org/abs/2005.14085`.

Section 1 is a quick introduction to inference systems with corules. Section 2 describes FJ and informally introduces our approach. In Section 3 we define coFJ and its abstract semantics, in Section 4 the operational semantics, in Section 5 we show some advanced examples, and in Section 6 we prove soundness. Finally, we discuss related work and draw conclusions in Section 7 and Section 8, respectively.

1 Inference systems with corules

First we recall standard notions on inference systems [1, 19]. Assuming a *universe* $\mathcal{U}$ of *judgments*, an *inference system* $\mathcal{I}$ is a set of *(inference) rules*, which are pairs $\frac{Pr}{c}$, with $Pr \subseteq \mathcal{U}$ the set of *premises*, and $c \in \mathcal{U}$ the *consequence* (a.k.a. *conclusion*). A rule with an empty set of premises is an *axiom*. A *proof tree* (a.k.a. *derivation*) for a judgment j is a tree whose nodes are (labeled with) judgments, j is the root, and there is a node c with children Pr only if there is a rule $\frac{Pr}{c}$.

The *inductive* and the *coinductive interpretation* of $\mathcal{I}$, denoted $Ind(\mathcal{I})$ and $CoInd(\mathcal{I})$, are the sets of judgments with, respectively, a finite[2], and a possibly infinite proof tree. In set-theoretic terms, let $F_{\mathcal{I}} : \wp(\mathcal{U}) \to \wp(\mathcal{U})$ be defined by $F_{\mathcal{I}}(S) = \{c \mid Pr \subseteq S, \frac{Pr}{c} \in \mathcal{I}\}$, and say that a set S is *closed* if $F_{\mathcal{I}}(S) \subseteq S$, *consistent* if $S \subseteq F_{\mathcal{I}}(S)$. Then, it can be proved that $Ind(\mathcal{I})$ is the smallest closed set, and $CoInd(\mathcal{I})$ is the largest consistent set. We write $\mathcal{I} \vdash j$ when j has a finite derivation in $\mathcal{I}$, that is, $j \in Ind(\mathcal{I})$.

An *inference system with corules*, or *generalized inference system*, is a pair $(\mathcal{I}, \mathcal{I}^{co})$ where $\mathcal{I}$ and $\mathcal{I}^{co}$ are inference systems, whose elements are called *rules* and *corules*, respectively. Corules can only be used in a special way, as defined below.

For a subset S of the universe, let $\mathcal{I}_{\sqcap S}$ denote the inference system obtained from $\mathcal{I}$ by keeping only rules with consequence in S. Let $(\mathcal{I}, \mathcal{I}^{co})$ be a generalized inference system. Then, its *interpretation* $Gen(\mathcal{I}, \mathcal{I}^{co})$ is defined by $Gen(\mathcal{I}, \mathcal{I}^{co}) = CoInd(\mathcal{I}_{\sqcap Ind(\mathcal{I} \cup \mathcal{I}^{co})})$.

In proof-theoretic terms, $Gen(\mathcal{I}, \mathcal{I}^{co})$ is the set of judgments that have a possibly infinite proof tree in $\mathcal{I}$, where all nodes have a finite proof tree in $\mathcal{I} \cup \mathcal{I}^{co}$, that is, the (standard) inference system consisting of rules and corules. We write $(\mathcal{I}, \mathcal{I}^{co}) \vdash j$ when j is derivable in $(\mathcal{I}, \mathcal{I}^{co})$, that is, $j \in Gen(\mathcal{I}, \mathcal{I}^{co})$. Note that $(\mathcal{I}, \emptyset) \vdash j$ is the same as $\mathcal{I} \vdash j$.

We illustrate these notions by a simple example. As usual, sets of rules are expressed by *meta-rules* with side conditions, and analogously sets of corules are expressed by *meta-corules* with side conditions. (Meta-)corules will be written with thicker lines, to be distinguished from (meta-)rules. The following inference system defines the minimum element of a list, where $[x]$ is the list consisting of only x, and $x : u$ the list with head x and tail u.

$$\frac{}{min([x], x)} \qquad \frac{min(u, y)}{min(x{:}u, z)}\, z = \min(x, y).$$

[2] Under the common assumption that sets of premises are finite, otherwise we should say well-founded.

The inductive interpretation gives the correct result only on finite lists, since for infinite lists an infinite proof is clearly needed. However, the coinductive one fails to be a function. For instance, for L the infinite list $2:1:2:1:2:1:\ldots$, any judgment $min(L,x)$ with $x \leq 1$ can be derived, as shown below.

$$\dfrac{\dfrac{\dfrac{\dots}{min(L,1)}}{min(1{:}L,1)}}{min(2{:}1{:}L,1)} \qquad \dfrac{\dfrac{\dfrac{\dots}{min(L,0)}}{min(1{:}L,0)}}{min(2{:}1{:}L,0)}$$

By adding a corule (in this case a coaxiom), wrong results are "filtered out":

$$\frac{}{min(x{:}\epsilon,x)} \qquad \frac{min(u,y)}{min(x{:}u,z)}\, z=\min(x,y) \qquad \genfrac{}{}{2pt}{}{}{min(x{:}u,x)}$$

Indeed, the judgment $min(2{:}1{:}L,1)$ has the infinite proof tree shown above, and each node has a finite proof tree in the inference system extended by the corule:

$$\dfrac{\dfrac{\dfrac{\dots}{min(L,1)}}{min(1{:}L,1)}}{min(2{:}1{:}L,1)} \qquad \dfrac{\genfrac{}{}{2pt}{}{}{min(1{:}L,1)}}{min(2{:}1{:}L,1)}$$

The judgment $min(2{:}1{:}L,0)$, instead, has the infinite proof tree shown above, but has *no finite proof tree* in the inference system extended by the corule. Indeed, since 0 does not belong to the list, the corule can never be applied. On the other hand, the judgment $min(L,2)$ has a finite proof tree with the corule, but cannot be derived since they it has no infinite proof tree. We refer to [4, 5, 6, 13] for other examples.

As final remark, note that requiring the existence of a finite proof tree with corules only for the root is not enough. For regular proof trees, the requirement to have such a proof tree for each node can be simplified in two ways:

- either requiring a sufficiently large finite proof-with-corules for the root, that is, a finite proof tree for the root which includes all the nodes of the regular proof tree
- or requiring a finite proof-with-corules for one node taken from each infinite path.

Let $(\mathcal{I},\mathcal{I}^{co})$ be a generalized inference system. The *bounded coinduction principle* [4], a generalization of the standard coinduction principle, can be used to prove *completeness* of $(\mathcal{I},\mathcal{I}^{co})$ w.r.t. a set S (for "specification") of *valid* judgments.

▶ **Theorem 1** (Bounded coinduction). *If the following two conditions hold:*

1. $S \subseteq Ind(\mathcal{I} \cup \mathcal{I}^{co})$, *that is, each valid judgment has a finite proof tree in* $\mathcal{I} \cup \mathcal{I}^{co}$;
2. $S \subseteq F_{\mathcal{I}}(S)$, *that is, each valid judgment is the consequence of a rule in* $\mathcal{I}$ *with premises in* S

then $S \subseteq Gen(\mathcal{I},\mathcal{I}^{co})$.

2 From FJ to coFJ

We recall FJ, and informally explain its extension with infinite objects and codefinitions.

Featherweight Java. The standard syntax and semantics in big-step style of FJ are shown in Figure 1. We omit cast since this feature does not add significant issues. We adopt a big-step, rather than a small-step style as in the original FJ definition, since in this way the semantics is directly defined by an inference system, denoted $\mathcal{I}_{\text{FJ}}$ in the following, which will be equipped with corules to support infinite objects. We write $\overline{cd}$ as metavariable for $cd_1 \ldots cd_n$, $n \geq 0$, and analogously for other sequences. We sometimes use the wildcard _ when the corresponding metavariable is not relevant.

$$
\begin{array}{llll}
cd & ::= & \texttt{class}\ C\ \texttt{extends}\ C'\ \{\ \overline{fd}\ \overline{md}\ \} & \text{class declaration} \\
fd & ::= & C\,f\,\texttt{;} & \text{field declaration} \\
md & ::= & C\ m(C_1\,x_1,\ldots,C_n\,x_n)\ \{e\} & \text{method declaration} \\
e \in \mathcal{E} & ::= & x \mid e.f \mid \texttt{new}\ C(\overline{e}) \mid e.m(\overline{e}) & \text{expression} \\
 & & & \\
v \in \mathcal{V} & ::= & \texttt{new}\ C(\overline{v}) & \text{(finite) object}
\end{array}
$$

$$
\text{(FJ-FIELD)}\ \frac{e \Downarrow v}{e.f \Downarrow v_i}\quad
\begin{array}{l} v = \texttt{new}\ C(v_1,\ldots,v_n) \\ \mathit{fields}(C) = f_1 \ldots f_n \\ f = f_i, i \in 1..n \end{array}
\qquad
\text{(FJ-NEW)}\ \frac{\overline{e} \Downarrow \overline{v}}{\texttt{new}\ C(\overline{e}) \Downarrow \texttt{new}\ C(\overline{v})}
$$

$$
\text{(FJ-INVK)}\ \frac{e_0 \Downarrow v_0 \quad \overline{e} \Downarrow \overline{v} \quad e[v_0/\texttt{this}][\overline{v}/\overline{x}] \Downarrow v}{e_0.m(\overline{e}) \Downarrow v}\quad
\begin{array}{l} v_0 = \texttt{new}\ C(\_) \\ \mathit{mbody}(C,m) = (\overline{x}, e) \end{array}
$$

Figure 1 FJ syntax and big-step rules.

A sequence of class declarations $\overline{cd}$ is called a *class table*. Each class has a canonical constructor whose parameters match the fields of the class, the inherited ones first. We assume standard FJ constraints, e.g., no field hiding and no method overloading. The only variables occurring in method bodies are parameters (including `this`). Values are *objects*, that is, constructor invocations where arguments are values in turn.

The judgment $e \Downarrow v$ is implicitly parameterized on a fixed class table. In the rules we use standard FJ auxiliary functions, omitting their formal definition. Notably, $\mathit{fields}(C)$ returns the sequence $f_1 \ldots f_n$ of the field names[3] of the class, in declaration order with the inherited first, and $\mathit{mbody}(C,m)$, for method m of the class, the pair of the sequence of parameters and the definition. Substitution $e[\overline{e}/\overline{x}]$, for $\overline{e}$ and $\overline{x}$ of the same length, is defined in the customary manner. Finally, for $\overline{e} = e_1 \ldots e_n$ and $\overline{v} = v_1 \ldots v_n$, $\overline{e} \Downarrow \overline{v}$ is an abbreviation for $e_1 \Downarrow v_1 \ldots e_n \Downarrow v_n$.

Rule (FJ-FIELD) models field access. If the selected field is actually a field of the receiver's class, then the corresponding value is returned as result. Rule (FJ-NEW) models object creation: if the argument expressions $\overline{e}$ evaluate to values $\overline{v}$, then the result is an object of class C. Rule (FJ-INVK) models method invocation. The receiver and argument expressions are evaluated first. Then, method look-up is performed, starting from the receiver's class, by the auxiliary function *mbody*. Lastly, the definition e of the method, where `this` is replaced by the receiver, and the parameters by the arguments, is evaluated, and its result is returned.

Infinite objects and codefinitions. We take as running example the following FJ implementation of lists of integers, equipped with some typical methods: `isEmpty` tests the emptiness, `incr` returns the list where all elements have been incremented by one, `allPos` checks whether all elements are positive, `member` checks whether the argument is in the list, and `min` returns the minimal element.

[3] We omit types since not relevant here. We discuss about type systems for coFJ in the conclusion.

```
class List extends Object {
  bool isEmpty() {true}
  List incr() {new EmptyList()}
  bool allPos() {true}
  bool member(int x) {false}
}
class EmptyList extends List { }
class NonEmptyList extends List {
  int head; List tail;
  bool isEmpty() {false}
  List incr() {new NonEmptyList(this.head+1,this.tail.incr())}
  bool allPos() {if (this.head<=0) false else this.tail.allPos()}
  bool member(int x) {if (this.head==x) true else this.tail.member(x)}
  int min() {
    if (this.tail.isEmpty()) this.head
    else Math.min(this.tail.min(),this.head)
  }
}
```

We used some additional standard constructs, such as conditional and primitive types `bool` and `int` with their operations; to avoid to use abstract methods, `List` provides the default implementation on empty lists, overridden in `NonEmptyList`, except for method `min` which is only defined on non empty lists.

In FJ we can represent finite lists. For instance, the object

```
new NonEmptyList(2, new NonEmptyList(1, new EmptyList()))
```

which we will abbreviate $[2, 1]$, represents a list of two elements, and it is easy to see that all the above method definitions provide the expected meaning on finite lists.

On the other hand, since the syntactic definition for objects is interpreted, like the others, inductively, in FJ objects are *finite*, hence we cannot represent, e.g., the infinite list of natural numbers $[0, 1, 2, 3, \ldots]$, abbreviated $[0..]$, or the infinite list $[2, 1, 2, 1, 2, 1, \ldots]$, abbreviated $[2, 1]^\omega$. To move from finite to infinite objects, it is enough to interpret the syntactic definition for values *coinductively*, so to obtain infinite terms as well. However, to make the extension significant, we should be able to *generate* such infinite objects as results of expressions, and to appropriately *handle* them by methods.

To generate infinite objects, e.g., the infinite lists mentioned above, a natural approach is to consider method definitions as *corecursive*, that is, to take the *coinductive* interpretation of the inference system in Figure 1. Consider the following class:

```
class ListFactory extends Object {
  NonEmptyList from(int x) {new NonEmptyList(x, this.from(x+1)}
  NonEmptyList two_one() {new NonEmptyList(2, this.one_two())}
  NonEmptyList one_two() {new NonEmptyList(1, this.two_one())}
}
```

With the standard FJ semantics, given by the inductive interpretation of the inference system in Figure 1, the method invocation `new ListFactory().from(0)` (abbreviated $\texttt{from}_0$ in the following) has no result, since there is no finite proof tree for a judgment of shape $\texttt{from}_0 \Downarrow \_$. Taking the coinductive interpretation, instead, such call returns as result the infinite list of natural numbers $[0..]$, since there is an infinite proof tree for the judgment $\texttt{from}_0 \Downarrow [0..]$. Analogously, the method invocation `new ListFactory().two_one()` returns $[2, 1]^\omega$. Moreover, the method invocations $[0..]$.`incr`() and $[2, 1]^\omega$.`incr`() correctly return as result the infinite lists $[1..]$ and $[3, 2]^\omega$, respectively.

However, in many cases to consider method definitions as corecursive is not satisfactory, since it leads to non-determinism, as shown for inference systems in Section 1. For instance, for the method invocation $[0..]$.`allPos()` both judgments $[0..]$.`allPos()` $\Downarrow$ `true` and $[0..]$.`allPos()` $\Downarrow$ `false` are derivable, and analogously for $[2,1]^\omega$.`allPos()`. In general, both results can be obtained for any infinite list of all positive numbers. A similar behavior is exhibited by method `member`: given an infinite list L which does not contain x, both judgments L.`member`(x) $\Downarrow$ `true` and L.`member`(x) $\Downarrow$ `false` are derivable. Finally, for the method invocation $[2,1]^\omega$.`min()`, any judgment $[2,1]^\omega$.`min()` $\Downarrow x$ with $x \leq 1$ can be derived.

To solve this problem, coFJ allows the programmer to *control* the semantics of corecursive methods by adding a *codefinition*[4], that is, an alternative method body playing a special role. Depending on the codefinition, the purely coinductive interpretation is refined, by filtering out some judgments. In the example, to achieve the expected meaning, the programmer should provide the following codefinitions.

```
class ListFactory extends Object {
  NonEmptyList from(int x) {
    new NonEmptyList(x, this.from(x+1)} corec {any}
  NonEmptyList one_two() {
    new NonEmptyList(1, this.two_one())} corec {any}
  NonEmptyList two_one() {
    new NonEmptyList(2, this.one_two())} corec {any}
}
class NonEmptyList extends List {
  int head; List tail;
  bool isEmpty() {false}
  List incr() {
    new NonEmptyList(this.head+1,this.tail.incr())} corec {any}
  bool allPos() {
    if (this.head <= 0) false else this.tail.allPos()} corec {true}
  bool member(int x) {
    if (this.head == x) true else this.tail.member(x)} corec {false}
  int min() {
    if (this.tail.isEmpty()) this.head
    else Math.min(this.tail.min(),this.head)
  } corec {this.head}
}
```

For the three methods of `ListFactory` and for the method `incr` the codefinition is `any`. This corresponds to keeping the coinductive interpretation as it is, as is appropriate in these cases since it provides only the expected result. In the other three methods, instead, the effect of the codefinition is to filter the results obtained by the coinductive interpretation. The way this is achieved is explained in the following section. Finally, for method `isEmpty` no codefinition is added, since the inductive behaviour works on infinite lists as well.

3 coFJ and its abstract semantics

We formally define coFJ, illustrate how the previous examples get the expected semantics, and show that, despite its non-determinism, coFJ is a conservative extension of FJ.

[4] The term "codefinition" is meant to suggest "alternative definition used to handle corecursion".

cd	$::=$	$\texttt{class}\ C\ \texttt{extends}\ C'\ \{\ \overline{fd}\ \overline{md}\ \}$	class declaration
fd	$::=$	$C\,f\texttt{;}$	field declaration
md	$::=$	$C\ m(C_1\,x_1,\ldots,C_n\,x_n)\ \{e\}\ [\texttt{corec}\ \{e'\}]$	method declaration with codefinition
$e \in \mathcal{E}$	$::=$	$x \mid e.f \mid \texttt{new}\ C(\overline{e}) \mid e.m(\overline{e})$	expression
$v \in \mathcal{V}^{\mathrm{a}}$	$::=_{\mathrm{CO}}$	$\texttt{new}\ C(\overline{v})$	possibly infinite object
$e \in \mathcal{E}^{\mathrm{a}}$	$::=$	$x \mid e.f \mid \texttt{new}\ C(\overline{e}) \mid e.m(\overline{e}) \mid v$	runtime expression

$$(\textsc{abs-field})\ \frac{e \Downarrow v}{e.f \Downarrow v_i}\ \begin{array}{l} v = \texttt{new}\ C(v_1,\ldots,v_n) \\ \mathit{fields}(C) = f_1 \ldots f_n \\ f = f_i, i \in 1..n \end{array} \qquad (\textsc{abs-new})\ \frac{\overline{e} \Downarrow \overline{v}}{\texttt{new}\ C(\overline{e}) \Downarrow \texttt{new}\ C(\overline{v})}$$

$$(\textsc{abs-invk})\ \frac{e_0 \Downarrow v_0 \quad \overline{e} \Downarrow \overline{v} \quad e[v_0/\texttt{this}][\overline{v}/\overline{x}] \Downarrow v}{e_0.m(\overline{e}) \Downarrow v}\ \begin{array}{l} v_0 = \texttt{new}\ C(\_) \\ \mathit{mbody}(C,m) = (\overline{x}, e) \end{array} \qquad (\textsc{abs-co-val})\ \frac{}{v \Downarrow v}$$

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

Figure 2 coFJ syntax and abstract semantics.

Formal definition of coFJ. The coFJ syntax is given in Figure 2. As the reader can note, the only difference is that method declarations include now, besides a definition e, an optional *codefinition* e', as denoted by the square brackets in the production. Furthermore, besides `this`, there is another special variable `any`, which can only occur in codefinitions. The codefinition will be used to provide an abstract semantics through an inference system with corules, where the role of `any` is to be a placeholder for an arbitrary value. For simplicity, we require the codefinition e' to be statically restricted to avoid recursive (even indirect) calls to the same method (we omit the standard formalization). Note that FJ is a (proper) subset of coFJ: indeed, an FJ class table is a coFJ class table with no codefinitions.

The syntactic definition for values is the same as before, but is now interpreted *coinductively*, as indicated by the symbol $::=_{\mathrm{CO}}$. In this way, infinite objects are supported. By replacing method parameters by arguments, we obtain *runtime expressions* admitting infinite objects as subterms. The sets $\mathcal{V}$ and $\mathcal{E}$ of FJ objects and expressions are subsets of $\mathcal{V}^{\mathrm{a}}$ and $\mathcal{E}^{\mathrm{a}}$, respectively. The judgment $e \Downarrow v$, with $e \in \mathcal{E}^{\mathrm{a}}$ and $v \in \mathcal{V}^{\mathrm{a}}$, is defined by an inference system with corules $(\mathcal{I}_{\mathrm{FJ}}, \mathcal{I}^{co}_{\mathrm{FJ}})$ where the rules $\mathcal{I}_{\mathrm{FJ}}$ are those[5] of FJ, as in Figure 1, and the corules $\mathcal{I}^{co}_{\mathrm{FJ}}$ are instances of two metacorules.

Corule (abs-co-val) is needed to obtain a value for infinite objects, as shown below. Corule (abs-co-invk) is analogous to the standard rule for method invocation, but uses the codefinition, and the variable `any` can be non-deterministically substituted with an arbitrary value. The auxiliary function *co-mbody* is defined analogously to *mbody*, but it returns the codefinition. Note that, even when $\mathit{mbody}(C,m)$ is defined, $\mathit{co\text{-}mbody}(C,m)$ can be undefined since no codefinition has been specified. This can be done to force a purely inductive behaviour for the method.

[5] To be precise, meta-rules are the same, with meta-variables e and v ranging on $\mathcal{E}^{\mathrm{a}}$, and $\mathcal{V}^{\mathrm{a}}$, respectively. However, we could have taken this larger universe in FJ as well without affecting the defined relation.

$$T_n = {\scriptstyle(\textsc{abs-invk})}\dfrac{{\scriptstyle(\textsc{abs-new})}\dfrac{}{\texttt{new LF()}\Downarrow\texttt{new LF()}} \quad {\scriptstyle(\textsc{n-val})}\dfrac{}{n\Downarrow n} \quad {\scriptstyle(\textsc{abs-new})}\dfrac{{\scriptstyle(\textsc{n-val})}\dfrac{}{n\Downarrow n} \qquad T_n}{\texttt{new NEL(}n\texttt{,new LF().from(}n\texttt{+1))}\Downarrow[n..]}}{\texttt{from}_n\Downarrow[n..]}$$

$$T_{n+1} = {\scriptstyle(\textsc{abs-invk})}\dfrac{{\scriptstyle(\textsc{abs-new})}\dfrac{}{\texttt{new LF()}\Downarrow\texttt{new LF()}} \quad {\scriptstyle(+)}\dfrac{\cdots}{n\texttt{+1}\Downarrow n\texttt{+1}} \quad {\scriptstyle(\textsc{abs-new})}\dfrac{{\scriptstyle(\textsc{n-val})}\dfrac{}{n\texttt{+1}\Downarrow n\texttt{+1}} \qquad T_{n+2}}{\texttt{new NEL(}n\texttt{+1,new LF().from(}n\texttt{+1+1))}\Downarrow[n\texttt{+1}..]}}{\texttt{new LF().from(}n\texttt{+1)}\Downarrow[n\texttt{+1}..]}$$

$${\scriptstyle(\textsc{abs-co-invk})}\genfrac{}{}{2pt}{0}{{\scriptstyle(\textsc{abs-new})}\dfrac{}{\texttt{new LF()}\Downarrow\texttt{new LF()}} \quad {\scriptstyle(\textsc{n-val})}\dfrac{}{n\Downarrow n} \quad {\scriptstyle(\textsc{abs-co-val})}\genfrac{}{}{2pt}{0}{}{[n..]\equiv\texttt{any}[\texttt{new LF()/this}][[n..]\texttt{/any}]\Downarrow[n..]}}{\texttt{from}_n\Downarrow[n..]}$$

Figure 3 Infinite (top) and finite (bottom) proof trees for $\texttt{from}_n \Downarrow [n..]$.

Examples. As an example, we illustrate in Figure 3 the role of the two corules for the call `new ListFactory().from(0)`. For brevity, we write abbreviated class names. Furthermore, $\texttt{from}_n$ stands for the call `new ListFactory().from(n)` and $[n..]$ for the infinite object `new NonEmptyList(n,new NonEmptyList(n+1,...)))`.

In the top part of Figure 3, we show the infinite proof tree T_n which can be constructed, for any natural number n, for the judgment $\texttt{from}_n \Downarrow [n..]$ without the use of corules. We use standard rules (n-val) and (+) to deal with integer constants and addition.

To derive the judgment in the inference system with corules, each node in this infinite tree should have a finite proof tree with the corules. Notably, this should hold for nodes of shape $\texttt{from}_n \Downarrow [n..]$, and indeed the finite proof tree for such nodes is shown in the bottom part of the figure. Note that, in this example, the result for the call $\texttt{from}_n$ is uniquely determined by the rules, hence the role of the corules is just to "validate" this result. To this end, the codefinition of the method `from` is the special variable `any`, which, when evaluating the codefinition, can be replaced by any value, hence, in particular, by the correct result $[n..]$. Corule (abs-co-val) is needed to obtain a finite proof tree for the infinite objects of shape $[n..]$. Analogous infinite and finite proof trees can be constructed for the judgments $\texttt{new ListFactory().two\_one()}\Downarrow[2,1]^\omega$, $[0..].\texttt{incr()}\Downarrow[1..]$ and $[2,1]^\omega.\texttt{incr()}\Downarrow[3,2]^\omega$.

For the method call $[0..].\texttt{allPos}()$, instead, both judgments $[0..].\texttt{allPos()}\Downarrow\texttt{true}$ and $[0..].\texttt{allPos()}\Downarrow\texttt{false}$ have an infinite proof tree. However, no finite proof tree using the codefinition can be constructed for the latter, whereas this is trivially possible for the former. Analogously, given an infinite list L which does not contain x, only the judgment $L.\texttt{member}(x)\Downarrow\texttt{false}$ has a finite proof tree using the codefinition.

Finally, for the method invocation $[2,1]^\omega.\texttt{min}()$, for any $v \leq 1$ there is an infinite proof tree built without corules for the judgment $[2,1]^\omega.\texttt{min()}\Downarrow v$ as shown in Figure 4. However, only the judgment $[2,1]^\omega.\texttt{min()}\Downarrow 1$ has a finite proof tree using the codefinition (Figure 5). For space reasons in both figures ellipses are used to omit the less interesting parts of the proof trees; we use the standard rule (if-f) for conditional, and the predefined function `Math.min` on integers.

Non-determinism and conservativity. The coFJ abstract semantics is inherently non-deterministic. Indeed, depending on the codefinition, the non-determinism of the coinductive interpretation may be kept. For instance, consider the following method declaration:

$$\text{(ABS-INVK)}\,\dfrac{T_0 \quad T_1}{[2,1]^\omega\texttt{.min()} \Downarrow v} \qquad T_0 = \text{(ABS-NEW)}\,\dfrac{\text{(N-VAL)}\,\dfrac{}{2\Downarrow 2} \quad \text{(ABS-NEW)}\,\dfrac{\text{(N-VAL)}\,\dfrac{}{1\Downarrow 1} \quad T_0}{[1,2]^\omega \Downarrow [1,2]^\omega}}{[2,1]^\omega \Downarrow [2,1]^\omega}$$

$$T_1 = \text{(IF-F)}\,\dfrac{\dfrac{\vdots}{[2,1]^\omega\texttt{.tail.isEmpty()}\Downarrow\texttt{false}} \quad \dfrac{\dfrac{\overset{T_2}{\vdots}}{[2,1]^\omega\texttt{.tail.min()}\Downarrow v} \quad \dfrac{\vdots}{[2,1]^\omega\texttt{.head}\Downarrow 2}}{\texttt{Math.min}([2,1]^\omega\texttt{.tail.min()}, [2,1]^\omega\texttt{.head})\Downarrow v}}{\textbf{if } [2,1]^\omega\texttt{.tail.isEmpty()} \textbf{ then } [2,1]^\omega\texttt{.head} \textbf{ else } \texttt{Math.min}([2,1]^\omega\texttt{.tail.min()}, [2,1]^\omega\texttt{.head})\Downarrow v}$$

$$T_2 = \text{(IF-F)}\,\dfrac{\dfrac{\vdots}{[1,2]^\omega\texttt{.tail.isEmpty()}\Downarrow\texttt{false}} \quad \dfrac{\dfrac{\overset{T_1}{\vdots}}{[1,2]^\omega\texttt{.tail.min()}\Downarrow v} \quad \dfrac{\vdots}{[1,2]^\omega\texttt{.head}\Downarrow 1}}{\texttt{Math.min}([1,2]^\omega\texttt{.tail.min()}, [1,2]^\omega\texttt{.head})\Downarrow v}}{\textbf{if } [1,2]^\omega\texttt{.tail.isEmpty()} \textbf{ then } [1,2]^\omega\texttt{.head} \textbf{ else } \texttt{Math.min}([1,2]^\omega\texttt{.tail.min()}, [1,2]^\omega\texttt{.head})\Downarrow v}$$

Figure 4 Infinite proof tree for $[2,1]^\omega\texttt{.min()}\Downarrow v$ with $v \leq 1$ (main tree at the top left corner).

$$\text{(ABS-INVK)}\,\dfrac{T_0 \quad \text{(IF-F)}\,\dfrac{\dfrac{\vdots}{[2,1]^\omega\texttt{.tail.isEmpty()}\Downarrow\texttt{false}} \quad \dfrac{\text{(ABS-CO-INVK)}\,\dfrac{\dots \quad \dfrac{\text{(ABS-CO-VAL)}\,\dfrac{}{[1,2]^\omega\Downarrow[1,2]^\omega}}{[1,2]^\omega\texttt{.head}\Downarrow 1}}{[2,1]^\omega\texttt{.tail.min()}\Downarrow 1} \quad \dfrac{\vdots}{[2,1]^\omega\texttt{.head}\Downarrow 2}}{\texttt{Math.min}([2,1]^\omega\texttt{.tail.min()}, [2,1]^\omega\texttt{.head})\Downarrow 1}}{\textbf{if } [2,1]^\omega\texttt{.tail.isEmpty()} \textbf{ then } [2,1]^\omega\texttt{.head} \textbf{ else } \texttt{Math.min}([2,1]^\omega\texttt{.tail.min()}, [2,1]^\omega\texttt{.head})\Downarrow 1}}{[2,1]^\omega\texttt{.min()}\Downarrow 1}$$

Figure 5 Finite proof tree with codefinition for $[2,1]^\omega\texttt{.min()}\Downarrow 1$ (T_0 as in Figure 4).

```
class C {
  C m() { this.m() } corec { any }
}
```

Method `m()` recursively calls itself. In the abstract semantics, the judgment `new C().m()`$\Downarrow v$ can be derived for any value v. In the operational semantics defined in Section 4, such method call evaluates to $(x, x : x)$, that is, the representation of *undetermined*.

However, determinism of FJ evaluation is preserved. Indeed, coFJ abstract semantics is a *conservative* extension of FJ semantics, as formally stated below.

▶ **Theorem 2** (Conservativity). *If* $\mathcal{I}_{FJ} \vdash e \Downarrow v$*, then* $(\mathcal{I}_{FJ}, \mathcal{I}^{co}_{FJ}) \vdash e \Downarrow v'$ *iff* $v = v'$.

Proof. Both directions can be easily proved by induction on the definition of $\mathcal{I}_{FJ} \vdash e \Downarrow v$. For the left-to-right direction, the fact that each syntactic category has a unique applicable meta-rule is crucial. ◀

This theorem states that, whichever the codefinitions chosen, coFJ does not change the semantics of expressions evaluating to some value in FJ. That is, coFJ abstract semantics allows derivation of new values only for expressions whose semantics is undefined in standard FJ, as in the examples shown above. Note also that, if no codefinition is specified, then the coFJ abstract semantics *coincides* with the FJ one, because corule (ABS-CO-INVK) cannot be applied, hence no infinite proof trees can be built for the evaluation of FJ expressions.

4 Operational semantics

We informally introduce the operational semantics of coFJ, provide its formal definition, and prove that it is deterministic and conservative.

Outline. In contrast to the abstract semantics of the previous section, the aim is to define a semantics which leads to an interpreter for the calculus. To obtain this, there are two issues to be considered:

1. infinite (regular) objects should be represented in a finite way;
2. infinite (regular) proof trees should be replaced by finite proof trees.

In the following we explain how these issues are handled in the coFJ operational semantics.

To obtain (1), we use an approach based on *capsules* [16], which are essentially expressions supporting cyclic references. In our context, capsules are pairs (e, σ) where e is an FJ expression and σ is an *environment*, that is, a finite mapping from variables into FJ expressions. Moreover, the following *capsule property* is satisfied: writing $FV(e)$ for the set of free variables in e, $FV(e) \subseteq dom(\sigma)$ and, for all $x \in dom(\sigma)$, $FV(\sigma(x)) \subseteq dom(\sigma)$. An FJ source expression e is represented by the capsule $(e, \varnothing)$, where $\varnothing$ denotes the empty environment. In particular, values are pairs (v, σ) where v is an *open* FJ object, that is, an object possibly containing variables. In this way, cyclic objects can be obtained: for instance, $(x, x : \texttt{new NEL}(2, \texttt{new NEL}(1, x)))$ represents the infinite regular list $[2,1]^\omega$ considered before.

To obtain (2), methods are *regularly corecursive*. This means that execution keeps track of the pending method calls, so that, when a call is encountered the second time, this is detected[6], avoiding non-termination as it would happen with ordinary recursion. Regular corecursion in coFJ is *flexible*, since the behaviour of the method when a cycle is detected is specified by the codefinition.

Consider, for instance, the method call `new ListFactory().two_one()`; thanks to regular corecursion, the result is the cyclic object $(x, x : \texttt{new NEL}(2, \texttt{new NEL}(1, x)))$. Indeed, the operational semantics associates a fresh variable, say, x, to the initial call, so that, when the same call is encountered the second time, the association $x : x$ is added in the environment, and the codefinition is evaluated where `any` is replaced by x. Hence, $(x, x : x)$ is returned as result, so that the result of the original call is $(x, x : \texttt{new NEL}(2, \texttt{new NEL}(1, x)))$. The call `new ListFactory().from(0)`, instead, does not terminate in the operational semantics, since no call is encountered more than once (the resulting infinite object is non-regular).

Consider now the call $[2,1]^\omega$`.allPos()`. In this case, when the call is encountered the second time, after an intermediate call $[1,2]^\omega$`.allPos()`, the result of the evaluation of the codefinition is `true`, so that the result of the original call is `true` as well.[7] If the codefinition were `any`, then the result would be $(x, x : x)$, that is, undetermined. Note that, if the list is finite, then no regular corecursion is involved, since the same call cannot occur more than once; the same holds 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, as $[2,1]^\omega$.

In the case of $[2,1]^\omega$`.min()`, when the call is encountered the second time the result of the evaluation of the codefinition is 2, so that the result of the intermediate call $[1,2]^\omega$`.min()` is 1, and this is also the result of the original call.

[6] The semantics detects an already encountered call by relying on capsule equivalence (Figure 7).
[7] To be rigorous, a capsule of shape (`true`, _).

Formal definition. To formally express the approach described above, the judgment of the operational semantics has shape $e, \sigma, \tau \Downarrow \mathrm{v}, \sigma'$ where: (e, σ) is the capsule to be evaluated; τ is a *call trace*, used to keep track of already encountered calls, that is, an injective map from *calls* $\mathrm{v}_0.m(\overline{\mathrm{v}})$ to (possibly tagged) variables, and (v, σ') is the capsule result. Variables in the codomain of the call trace have a tag `ck` during the checking step for the corresponding call, as detailed below. The pair (e, σ) and (v, σ') are assumed to satisfy the capsule property.

The semantic rules are given in Figure 6. We denote by $\sigma\{x{:}\mathrm{v}\}$ the environment which gives v on x, and is equal to σ elsewhere, and analogously for other maps. Furthermore, we use the following notations, formally defined in Figure 7.

- $\mathit{unfold}(\mathrm{v}, \sigma)$ is the *unfolding* of v in σ, that is, the corresponding object, if any.
- $\sigma_1 \sqcup \sigma_2$ is the *union of environments*, defined if they agree on the common domain.
- $(\mathrm{v}, \sigma) \approx (\mathrm{v}', \sigma')$ is the *equivalence of capsules.* As will be formalized in the first part of Section 6, equivalent capsules denote the same sets of abstract objects. This equivalence is extended by congruence to expressions, in particular to calls $\mathrm{v}_0.m(\overline{\mathrm{v}})$.
- $\tau_{\approx\sigma}$ is obtained by extending τ *up to equivalence* in σ. That is, detection of already encountered calls is performed up-to equivalence in the current environment.

Rule (val) is needed for objects which are not FJ objects. Rule (field) is similar to that of FJ except that the capsule (v, σ') must be unfolded to retrieve the corresponding object. Furthermore, the resulting environment is that obtained by evaluating the receiver. Rule (new) is analogous to that of FJ. The resulting environment is the union of those obtained by evaluating the arguments.

There are four rules for method invocation. In all of them, as in the FJ rule, the receiver and argument expressions are evaluated first to obtain the call $c = \mathrm{v}.m(\overline{\mathrm{v}})$. The environment $\widehat{\sigma}$ is the union of those obtained by these evaluations. Then, the behavior is different depending whether such call (meaning a call equivalent to c in $\widehat{\sigma}$) has been already encountered.

Rules (invk-ok) and (invk-check) handle[8] a call c which is encountered the first time, as expressed by the side condition $c \notin \mathit{dom}(\tau_{\approx\widehat{\sigma}})$. In both, the definition e, where the receiver replaces `this` and the arguments replace the parameters, is evaluated. Such evaluation is performed in the call trace τ updated to associate the call c with an unused variable x (in these two rules "x fresh" means that x does not occur in the derivations of $e_i, \sigma, \tau \Downarrow \mathrm{v}_i, \sigma_i'$, for all $i \in 0..n$), and produces the capsule (v, σ'). Then there are two cases, depending on whether $x \in \mathit{dom}(\sigma')$ holds.

If $x \notin \mathit{dom}(\sigma')$, then the evaluation of the definition for c has been performed without evaluating the codefinition. That is, the same call has not been encountered, hence the result has been obtained by standard recursion, and no additional check is needed.

If $x \in \mathit{dom}(\sigma')$, instead, then the evaluation of the definition for c has required to evaluate the codefinition. In this case, an additional check is required (third premise). That is, $e[\mathrm{v}_0/\mathtt{this}][\overline{\mathrm{v}}/\overline{x}]$ is evaluated once more under the assumption that v is the result of the call. Formally, evaluation takes place in an environment updated to associate x with v, and the variable x corresponding to the call is tagged with `ck`. The capsule result obtained in this way must be (equivalent to) that obtained by the first evaluation of the body of the method. In Section 5 we discuss in detail the role of this additional check, showing an example where it is necessary. If the check succeeds, then the final result is the variable x in the environment updated to associate x with v. Otherwise, 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 is different from the expected one, as should be done in an implementation.

[8] The two rules could be merged together, but we prefer to make explicit the difference for sake of clarity.

$$
\begin{array}{llll}
\mathrm{v} \in \mathcal{V}^{\mathrm{op}} & ::= & \texttt{new}\ C(\overline{\mathrm{v}}) \mid x & \text{open object} \\
\sigma & ::= & x_1 : \mathrm{v}_1 \ldots\ x_n : \mathrm{v}_n \quad (n \geq 0) & \text{environment} \\
c & ::= & \mathrm{v}.m(\overline{\mathrm{v}}) & \text{call} \\
t & ::= & [\mathsf{ck}] & \text{optional checking tag} \\
\tau & ::= & c_1 : x_1^{t_1}, \ldots, c_n : x_n^{t_n} \quad (n \geq 0) & \text{call trace}
\end{array}
$$

$$
\text{(VAL)}\ \frac{}{\mathrm{v}, \sigma, \tau \Downarrow \mathrm{v}, \sigma}
\qquad
\text{(FIELD)}\ \frac{e, \sigma, \tau \Downarrow \mathrm{v}, \sigma'}{e.f, \sigma, \tau \Downarrow \mathrm{v}_i, \sigma'}
\quad
\begin{array}{l}
\mathit{unfold}(\mathrm{v}, \sigma') = \texttt{new}\ C(\mathrm{v}_1, \ldots, \mathrm{v}_n) \\
\mathit{fields}(C) = f_1 \ldots f_n \\
f = f_i, i \in 1..n
\end{array}
$$

$$
\text{(NEW)}\ \frac{e_i, \sigma, \tau \Downarrow \mathrm{v}_i, \sigma'_i \quad \forall i \in 1..n}{\texttt{new}\ C(e_1, \ldots, e_n), \sigma, \tau \Downarrow \texttt{new}\ C(\mathrm{v}_1, \ldots, \mathrm{v}_n), \bigsqcup_{i \in 1..n} \sigma'_i}
$$

In all the following rules:
$$
\begin{array}{l}
\overline{e} = e_1, \ldots, e_n \\
\overline{\mathrm{v}} = \mathrm{v}_1 \ldots \mathrm{v}_n \\
c = \mathrm{v}_0.m(\overline{\mathrm{v}}) \\
\widehat{\sigma} = \bigsqcup_{i \in 0..n} \sigma'_i \\
\mathit{unfold}(\mathrm{v}_0, \sigma'_0) = \texttt{new}\ C(\_)
\end{array}
$$

$$
\text{(INVK-OK)}\ \frac{\begin{array}{l} e_i, \sigma, \tau \Downarrow \mathrm{v}_i, \sigma'_i \quad \forall i \in 0..n \\ e[\mathrm{v}_0/\texttt{this}][\overline{\mathrm{v}}/\overline{x}], \widehat{\sigma}, \tau\{c : x\} \Downarrow \mathrm{v}, \sigma' \end{array}}{e_0.m(\overline{e}), \sigma, \tau \Downarrow \mathrm{v}, \sigma'}
\quad
\begin{array}{l}
c \notin \mathit{dom}(\tau_{\approx \widehat{\sigma}}) \\
x \text{ fresh} \\
\mathit{mbody}(C, m) = (\overline{x}, e) \\
x \notin \mathit{dom}(\sigma')
\end{array}
$$

$$
\text{(INVK-CHECK)}\ \frac{\begin{array}{l} e_i, \sigma, \tau \Downarrow \mathrm{v}_i, \sigma'_i \quad \forall i \in 0..n \\ e[\mathrm{v}_0/\texttt{this}][\overline{\mathrm{v}}/\overline{x}], \widehat{\sigma}, \tau\{c : x\} \Downarrow \mathrm{v}, \sigma' \\ e[\mathrm{v}_0/\texttt{this}][\overline{\mathrm{v}}/\overline{x}], \widehat{\sigma} \sqcup \sigma'\{x : \mathrm{v}\}, \tau\{c : x^{\mathsf{ck}}\} \Downarrow \mathrm{v}', \sigma'' \end{array}}{e_0.m(\overline{e}), \sigma, \tau \Downarrow x, \sigma'\{x : \mathrm{v}\}}
\quad
\begin{array}{l}
c \notin \mathit{dom}(\tau_{\approx \widehat{\sigma}}) \\
x \text{ fresh} \\
\mathit{mbody}(C, m) = (\overline{x}, e) \\
x \in \mathit{dom}(\sigma') \\
(x, \sigma'\{x : \mathrm{v}\}) \approx (\mathrm{v}', \sigma'')
\end{array}
$$

$$
\text{(COREC)}\ \frac{\begin{array}{l} e_i, \sigma, \tau \Downarrow \mathrm{v}_i, \sigma'_i \quad \forall i \in 0..n \\ e'[\mathrm{v}_0/\texttt{this}][\overline{\mathrm{v}}/\overline{x}][x/\texttt{any}], \widehat{\sigma}\{x : x\}, \tau \Downarrow \mathrm{v}, \sigma' \end{array}}{e_0.m(\overline{e}), \sigma, \tau \Downarrow \mathrm{v}, \sigma'\{x : x\}}
\quad
\begin{array}{l}
\tau_{\approx \widehat{\sigma}}(c) = x \\
\mathit{co\text{-}mbody}(C, m) = (\overline{x}, e')
\end{array}
$$

$$
\text{(LOOK-UP)}\ \frac{e_i, \sigma, \tau \Downarrow \mathrm{v}_i, \sigma'_i \quad \forall i \in 0..n}{e_0.m(\overline{e}), \sigma, \tau \Downarrow x, \widehat{\sigma}}
\quad
\tau_{\approx \widehat{\sigma}}(c) = x^{\mathsf{ck}}
$$

Figure 6 coFJ operational semantics.

$$
\begin{aligned}
&\mathit{unfold}(\mathrm{v},\sigma) = \begin{cases} \mathtt{new}\ C(\overline{\mathrm{v}}) & \text{if } \mathrm{v} = \mathtt{new}\ C(\overline{\mathrm{v}}) \\ \mathit{unfold}(\sigma(\mathrm{v}),\sigma) & \text{if } \mathrm{v} = x \end{cases}\\
&\mathit{undet}(\sigma) = \{x \in \mathit{dom}(\sigma) \mid \mathit{unfold}(x,\sigma)\uparrow\}
\end{aligned}
$$

For σ_1 and σ_2 such that $\sigma_1(x) = \sigma_2(x)$ for all $x \in \mathit{dom}(\sigma_1) \cap \mathit{dom}(\sigma_2)$

$$
(\sigma_1 \sqcup \sigma_2)(x) = \begin{cases} \sigma_1(x) & x \in \mathit{dom}(\sigma_1) \\ \sigma_2(x) & x \in \mathit{dom}(\sigma_2) \end{cases}
$$

Set $\overset{\sigma}{\leftrightarrow}$ the least equivalence relation on $\mathit{undet}(\sigma)$ such that $x \overset{\sigma}{\leftrightarrow} y$ if $\sigma(x) = y$, $[x]$ the equivalence class of x, and $\mathit{undet}_{\leftrightarrow}(\sigma)$ the quotient. A relation $\alpha \subseteq \mathit{undet}(\sigma_1) \times \mathit{undet}(\sigma_2)$ is a *σ_1,σ_2-renaming* if it induces a (partial) bijection from $\mathit{undet}_{\leftrightarrow}(\sigma_1)$, still denoted α, to $\mathit{undet}_{\leftrightarrow}(\sigma_2)$. Given α a σ_1,σ_2-renaming, the relation $(x,\sigma_1)\approx_\alpha(x',\sigma_2)$ is coinductively defined by:

$$
\frac{}{(x,\sigma)\approx_\alpha(x',\sigma')}\; x\alpha x' \qquad \frac{(\mathrm{v}_i,\sigma)\approx_\alpha(\mathrm{v}'_i,\sigma') \quad \forall i \in 1..n}{(\mathrm{v},\sigma)\approx_\alpha(\mathrm{v}',\sigma')}\; \begin{array}{l} \mathit{unfold}(\mathrm{v},\sigma) = \mathtt{new}\ C(\mathrm{v}_1,..,\mathrm{v}_n) \\ \mathit{unfold}(\mathrm{v}',\sigma') = \mathtt{new}\ C(\mathrm{v}'_1,..,\mathrm{v}'_n) \end{array}
$$

A σ_1,σ_2-renaming α is *strict* if, for $x,y \in \mathit{undet}(\sigma_1) \cap \mathit{undet}(\sigma_2)$, $[x]\alpha[y]$ iff $x \overset{\sigma_1}{\leftrightarrow} y$ and $x \overset{\sigma_2}{\leftrightarrow} y$.
We write $(\mathrm{v},\sigma)\approx(\mathrm{v}',\sigma')$ if $(\mathrm{v},\sigma)\approx_\alpha(\mathrm{v}',\sigma')$ for some strict α.

$\tau_{\approx\sigma}(c') = \tau(c)$ for each c' such that $(c',\sigma)\approx(c,\sigma)$

Figure 7 coFJ auxiliary definitions.

The remaining rules handle an already encountered call c, that is, $\tau_{\approx\widehat{\sigma}}(c)$ is defined. The behaviour is different depending on whether the corresponding variable x is tagged or not.

If x is not tagged, then rule (COREC) evaluates the codefinition where the receiver object replaces `this`, the arguments replace the parameters, and, furthermore, the variable x found in the call trace replaces `any`. In addition, $\widehat{\sigma}$ is updated to associate x with x. In this way, the semantics keeps track of the application of rule (COREC).

If x is tagged, instead, then we are in a checking step for the corresponding call. In this case, rule (LOOK-UP) simply returns the associated variable for a call; by definition of the operational semantics, in this case such a variable is always defined in the environment.

Figure 7 contains the formal definitions of the notations used in the rules.

Note that *unfold*, being inductively defined, can be undefined, denoted $\uparrow$, in presence of unguarded cycles among variables. Capsule equivalence, instead, is defined coinductively, so that, e.g., $(x, x\!:\!\mathtt{new}\ C(x))$ is equivalent to $(x, x\!:\!\mathtt{new}\ C(\mathtt{new}\ C(x)))$. Capsule equivalence implicitly subsumes α-equivalence of variables whose unfolding is defined, e.g., $(x, x\!:\!\mathtt{new}\ C(x))$ is equivalent to $(y, y:\mathtt{new}\ C(y))$. Instead, α-equivalence of undetermined variables is given by an explicit renaming, which should preserve disjointness of cycles. For instance, $(\mathtt{new}\ C(x,y),(x:y,y:x))$ is equivalent to $(\mathtt{new}\ C(x,x), x:x)$, but is *not* equivalent to $(\mathtt{new}\ C(x,y),(x:x,y:y))$. Indeed, in the latter case x and y can be instantiated independently. We will prove in Section 6 (Theorem 10) that the relation $\approx_\alpha$, for some σ_1,σ_2-renaming α, is the operational counterpart of the fact that two capsules denote the same set of abstract values. The stronger strictness condition prevents erroneous identification of objects during evaluation, e.g., $(\mathtt{new}\ C(x,y),(x:x,y:y))$ is not equivalent to $(\mathtt{new}\ C(y,x),(y:y,x:x))$.

Determinism and conservativity. In contrast to coFJ abstract semantics, but like FJ, coFJ operational semantics is deterministic.

▶ **Theorem 3** (Determinism). *If $e, \sigma, \tau_1 \Downarrow \mathrm{v}_1, \sigma_1$ and $e, \sigma, \tau_2 \Downarrow \mathrm{v}_2, \sigma_2$ hold and $dom(\tau_1) = dom(\tau_2)$, then (v_1, σ_1) and (v_2, σ_2) are equal up-to α-equivalence.*

Proof. The proof is by induction on the derivation for $e, \sigma, \tau_1 \Downarrow \mathrm{v}_1, \sigma_1$. The key point is that, once fixed e, σ and $dom(\tau_1)$, there is a unique applicable rule, hence both $e, \sigma, \tau_1 \Downarrow \mathrm{v}_1, \sigma_1$ and $e, \sigma, \tau_2 \Downarrow \mathrm{v}_2, \sigma_2$ are derived by the same rule. ◀

As the abstract one, the operational semantics is a conservative extension of the standard FJ semantics. This result follows from soundness with respect to the abstract semantics in next section, however the direct proof below provides some useful insight.

▶ **Theorem 4** (Conservativity). *If $\mathcal{I}_{\text{FJ}} \vdash e \Downarrow v$, then $e, \varnothing, \varnothing \Downarrow \mathrm{v}, \sigma$ holds iff $\mathrm{v} = v$ and $\sigma = \varnothing$.*

For the proof, we need some auxiliary lemmas and definitions. First, we note that FJ has the *strong determinism* property: each expression has at most one finite proof tree in $\mathcal{I}_{\text{FJ}}$.

▶ **Lemma 5** (FJ strong determinism). *If $\mathcal{I}_{\text{FJ}} \vdash e \Downarrow v_1$ by a proof tree t_1 and $\mathcal{I}_{\text{FJ}} \vdash e \Downarrow v_2$ by a proof tree t_2, then $t_1 = t_2$ and $v_1 = v_2$.*

Proof. By induction on the definition of $e \Downarrow v_1$. The key point is that each judgement is the consequence of exactly one rule. ◀

By relying on strong determinism, it is easy to see that in FJ a proof tree for an expression cannot contain another node labelled by the same expression. In other words, if the evaluation of e requires to evaluate e again, then the FJ semantics is undefined on e, as expected.

▶ **Lemma 6.** *A proof tree in $\mathcal{I}_{\text{FJ}}$ for $e \Downarrow v$ cannot contain any other node $e \Downarrow v'$, for any v'.*

Proof. By Lemma 5, there is a unique proof tree t for the expression e. Hence, a node $e \Downarrow v'$ in t would be necessarily the root of a subtree of t equal to t, that is, it is the root of t. ◀

▶ **Definition 7.** *Let $\mathcal{I}_{\text{FJ}} \vdash e \Downarrow v$. A call trace τ is* disjoint *from $e \Downarrow v$ if in its proof tree*[9] *there are no instances of* (FJ-INVK) *where $v_0.m(\overline{v}) \in dom(\tau)$.*

▶ **Lemma 8.** *If $\mathcal{I}_{\text{FJ}} \vdash e \Downarrow v$, then, for all τ disjoint from $e \Downarrow v$, we have $e, \varnothing, \tau \Downarrow v, \varnothing$.*

Proof. The proof is by induction on the definition of $e \Downarrow v$.

(FJ-field) Let τ be a call trace disjoint from $e.f \Downarrow v_i$. Since $\mathcal{I}_{\text{FJ}} \vdash e \Downarrow v$, with $v = \texttt{new}\ C(v_1, \dots, v_n)$, holds by hypothesis, and τ is, by definition, also disjoint from $e \Downarrow v$, we get $e, \varnothing, \tau \Downarrow v, \varnothing$ by induction hypothesis. Then, since $unfold(v, \varnothing) = v$, we get $e.f, \varnothing, \tau \Downarrow v_i, \varnothing$ by rule (FIELD).

(FJ-new) Let τ be a call trace disjoint from $\texttt{new}\ C(e_1, \dots, e_n) \Downarrow \texttt{new}\ C(v_1, \dots, v_n)$. For all $i \in 1..n$, since $\mathcal{I}_{\text{FJ}} \vdash e_1 \Downarrow v_i$ holds by hypothesis, and τ is, by definition, also disjoint from $e_i \Downarrow v_i$, we get $e_i, \varnothing, \tau \Downarrow v_i, \varnothing$ by induction hypothesis. Then, we get $\texttt{new}\ C(e_1, \dots, e_n), \varnothing, \tau \Downarrow \texttt{new}\ C(v_1, \dots, v_n), \varnothing$ by rule (NEW).

(FJ-invk) Let τ be a call trace disjoint from $e_0.m(e_1, \dots, e_n) \Downarrow v$. For all $i \in 0..n$, since $\mathcal{I}_{\text{FJ}} \vdash e_i \Downarrow v_i$ holds by hypothesis, and τ is, by definition, also disjoint from $e_i \Downarrow v_i$, we get $e_i, \varnothing, \tau \Downarrow v_i, \varnothing$ by induction hypothesis. Set $\overline{v} = v_1 \dots v_n$ and $e' = e[v_0/\texttt{this}][\overline{v}/\overline{x}]$. By hypothesis, $\mathcal{I}_{\text{FJ}} \vdash e' \Downarrow v$ and, by definition, τ is also disjoint from $e' \Downarrow v$; furthermore, by Lemma 6, e' cannot occur twice in the proof tree for $e' \Downarrow v$, hence $\tau\{v_0.m(\overline{v}) : x\}$ is disjoint from $e' \Downarrow v$, for any fresh variable x. Then, by induction hypothesis, we have $e', \varnothing, \tau\{v_0.m(\overline{v}) : x\} \Downarrow v, \varnothing$, thus we get $e_0.m(e_1, \dots, e_n), \varnothing, \tau \Downarrow v, \varnothing$ by rule (INVK-OK). ◀

[9] Unique thanks to Lemma 5.

We can now prove the conservativity result for coFJ operational semantics.

Proof of Theorem 4. The right-to-left direction follows from Lemma 8, since $\varnothing$ is disjoint from any expression, while the other direction follows from the right-to-left one and Theorem 3. ◀

For coFJ operational semantics we can prove an additional result, characterizing derivable judgements which produce an empty environment. The meaning is that all results obtained *without using the codefinitions* are original FJ results.

▶ **Lemma 9.** *If* $e, \varnothing, \tau \Downarrow \mathrm{v}, \varnothing$ *holds, then* v *is an FJ value* v*, and* $\mathcal{I}_{FJ} \vdash e \Downarrow v$.

5 Advanced examples

This section provides some more complex examples to better understand the operational semantics of coFJ in Section 4 and its relationship with the abstract semantics in Section 3.

Examples on lists. We first show an example motivating the additional checking step (third premise) in rule (INVK-CHECK). Essentially, the success of this check for some capsule result corresponds to the existence of an infinite tree in the abstract semantics, whereas the fact that this capsule result is obtained by assuming the codefinition as result of the cyclic call (second premise) corresponds to the existence of a finite tree which uses the codefinition .

Assume to add to our running example of lists of integers a method that returns the sum of the elements. For infinite regular lists, that is, lists ending with a cycle, a result should be returned if the cycle has sum 0, for instance for a list ending with infinitely many 0s, and no result if the cycle has sum different from 0. This can be achieved as follows.

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

It is easy to see that the abstract semantics of the previous section formalizes the expected behavior. For instance, an infinite tree for a judgment $[2,1]^\omega$.`sum()` $\Downarrow v$ only exists for $v = 2 + 1 + v$, and there are no solutions of this equation, hence there is no result. In the operational semantics, by evaluating the body assuming the codefinition as result of the cyclic call (second premise of rule (INVK-CHECK)) the spurious result 3 would be returned. This is avoided by the third premise, which evaluates the method body assuming 3 as result of the cyclic call. Since we *do not* get 3 in turn as result, evaluation is stuck, as expected.

Note that the stuckness situation is detected: the last side-condition of rule (INVK-CHECK) fails, and a dynamic error (not modeled for simplicity, see the comments to the rule) is raised, likely an exception in an implementation. On the other hand, computations which *never* reach (a base case or) an already encountered call still do not terminate in this operational semantics, exactly as in the standard one, and the fact that this does not happen should be *proved* by suitable techniques, see the Conclusion.

All the examples shown until now have a constant codefinition. We show now an example where this is not enough. Consider the method `remPos()` that removes positive elements. A first attempt at a coFJ definition is the following:

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

Is this definition correct? Actually, it provides the expected behavior on finite lists, and cyclic lists where the cycle contains only positive elements. However, when the cycle contains at least one non positive element, there is no result. For instance, consider the method call $[0,1]^\omega$`.remPos()`. In the abstract semantics, an infinite tree can be constructed for the judgment $[0,1]^\omega$`.remPos()`$\Downarrow v$ only if $v = 0 : v$, and this clearly only holds for $v = [0]^\omega$. However, no finite tree can be constructed for this judgment using the codefinition. Note that, in the operational semantics, without the additional check (third premise of rule (INVK-CHECK)), we would get the spurious result $[0]$. In order to have a coFJ definition complete with respect to the expected behavior, we should provide a different codefinition for lists with infinitely many non-positive elements.

```
class NonEmptyList extends List { ...
  List remPos() {
    if(this.head > 0) this.tail.remPos()
    else new NonEmptyList(this.head,this.tail.remPos())}
  corec {if (this.allPos() then new EmptyList() else any}
```

Arithmetic with rational and real numbers. All real numbers in the closed interval $\{0..1\}$ can be represented by infinite lists $[d_1, d_2, \ldots]$ of decimal digits; more precisely, the infinite list $[d_1, d_2, \ldots]$ represents the real number which is the limit of the series $\sum_{i=1}^{\infty} 10^{-i} d_i$.

It is well-known that all rational numbers in $\{0..1\}$ correspond to either a terminating or repeating decimal, hence they can be represented by infinite regular lists of digits, where terminating decimals end with either an infinite sequence of 0 or an infinite sequence of 9; for instance, the terminating decimal $\frac{1}{2}$ can be represented equivalently by either $[5, 0, 0, \ldots]$ or $[4, 9, 9, \ldots]$, while the repeating decimal $\frac{1}{3}$ is represented by $[3, 3, \ldots]$.

Therefore, in coFJ all rational numbers in $\{0..1\}$ can be effectively represented with infinite precision at the level of the operational semantics; to this aim, we can declare a class `Number` with the two fields `digit` of type `int` and `others` of type `Number`: `digit` contains the leftmost digit, that is, the most significant, while `others` refers to the remaining digits, that is, the number we would obtain by a single left shift (corresponding to multiplication by 10). Since also non-regular values are allowed, in the abstract semantics class `Number` can be used to represent also all irrational numbers in $\{0..1\}$.

We now show how it is possible to compute in coFJ the addition of rational numbers in $\{0..1\}$ with infinite precision. We first define the method `carry` which computes the carry of the addition of two numbers: its result is 0 if the sum belongs to $\{0..1\}$, 1 otherwise.

```
class Number extends Object { // numbers in {0..1}
  int digit; // leftmost digit
  Number others; // all other digits

  int carry(Number num){ // returns 0 if this+num<=1, 1 otherwise
    if (this.digit+num.digit!=9) (this.digit+num.digit)/10
    else this.others.carry(num.others)
  } corec {0}
}
```

The two numbers `this` and `num` are inspected starting from the most significant digits: if their sum is different from 9, then the carry can be computed without inspecting the other digits, hence the integer division by 10 of the sum is returned. Corecursion is needed when the sum of the two digits equals 9; in this case the carry is the same obtained from the addition of `this.others` and `num.others`.

Finally, in the codefinition the carry 0 is returned; indeed, the codefinition is evaluated only when the sum of the digits for all positions inspected so far is 9 and the same patterns of digits are encountered for the second time. This can only happen for pairs of numbers whose addition is $[9, 9, \ldots]$, that is, 1, hence the computed carry must be 0.

Based on method `carry`, we can define method `add` which computes the addition of two numbers, excluding the possible carry in case of overflow.

```
class Number extends Object { ... // declarations as above
  Number add(Number num){ // returns this+num
    new Number(
      (this.digit+num.digit+this.others.carry(num.others))%10,
      this.others.add(num.others))} corec {any}
}
```

For each position, the corresponding digits of `this` and `num` are added to the carry computed for the other digits (`this.others.carry(num.others)`), then the reminder of the division by 10 gives the most significant digit of the result, whereas the others are obtained by corecursively calling the method on the remaining digits (`this.others.add(num.others)`). Since this call is guarded by a constructor call, the codefinition is `any`.

Note that, in the abstract semantics, methods `carry` and `add` correctly work also for irrational numbers.

Method `add` above is simple, but has the drawback that the same carries are computed more times; hence, in the worst case, the time complexity is quadratic in the period[10] of the two involved repeating decimals. To overcome this issue, we present a more elaborate example where carries are computed only once for any position; this is achieved by method `all_carries` below, which returns the sequence of all carries (hence, a list of binary digits).

Method `simple_add` corecursively adds all digits without considering carries, while method `add`, defined on top of `simple_add` and `all_carries`, computes the final result. This new version of `add` is not recursive and, hence. does not need a codefinition.

```
class Number extends Object { ... // declarations as above
  Number all_carries(Number num){ // carries for all positions
    this.simple_carries(num).complete()
  }
  Number simple_carries(Number num){ // carries computed immediately
    if(this.digit+num.digit!=9)
      new Number((this.digit+num.digit)/10,
        this.others.simple_carries(num.others))
    else new Number(9,this.others.simple_carries(num.others))
  } corec {any}

  Number complete(){ // computes missing carries marked with 9
    if(this.digit!=9) new Number(this.digit,this.others.complete())
    else this.fill(this.carry_lookahead()).complete()
  } corec {any}
```

[10] Indeed, the worst case scenario is when the carry propagates over all digits because their sum is always 9, and this can happen only if the two numbers have the same period.

```
  Number fill(int dig){ // fills with dig all next missing carries
    if(this.digit!=9) this else new Number(dig,this.others.fill(dig))
  } corec {any}

  int carry_lookahead(){ // returns the next computed carry
    if(this.digit!=9) this.digit else this.others.carry_lookahead()
  } corec {0}

  Number simple_add(Number num){ // addition without carries
    new Number((this.digit+num.digit)%10,
      this.others.simple_add(num.others))
  } corec {any}

  Number add(Number num){
    this.simple_add(num).simple_add(this.all_carries(num).others)
  }
}
```

Distances on graphs. The last example of this section involves graphs, which are the paradigmatic example of cyclic data structure. Our aim is to compute the *distance*, that is, the minimal length of a path, between two vertexes[11]. Consider a graph (V, adj) where V is the set of vertexes and $adj : V \to \wp(V)$ gives, for each vertex, the set of the adjacent vertexes. Each vertex has an identifier `id` assumed to be unique. We assume a class `Nat`$^\infty$, with subclasses `Nat` with an integer field, and `Infty` with no fields, for naturals and ∞ (distance between unconnected nodes), respectively. Such classes offer methods `succ()` for the successor, and `min(Nat`$^\infty$ `n)` for the minimum, with the expected behaviour (e.g., `succ` in class `Nat`$^\infty$ returns ∞).

```
class Vertex extends Object {
  Id id; AdjList adjVerts;
  Nat∞dist(Id id) {
    this.id==id?new Nat(0):this.adjVerts.dist(id).succ()}
  corec {new Infty()}
}

class AdjList extends Object { }
class EAdjList extends AdjList {
  Nat∞dist(Id id) { new Infty() }
}
class NEAdjList extends AdjList {
  Vertex vert; AdjList adjVerts;
  Nat∞dist(Id id) {this.vert.dist(id).min(this.adjVerts.dist(id))}
}
```

Clearly, if the destination `id` and the source node coincide, then the distance is 0. Otherwise, the distance is obtained by incrementing by one the minimal distance from an adjacent to `id`, computed by method `dist()` of `AdjList` called on the adjacency list. The codefinition of method `dist()` of class `Vertex` is needed since, in presence of a cycle, ∞ is returned and non-termination is avoided. The same approach can be adopted for visiting a graph: instead of keeping trace of already encountered nodes, cycles are implicitly handled by the loop detection mechanism of coFJ.

[11] The example can be easily adapted to weighted paths.

6 Soundness

Soundness of the operational semantics with respect to the abstract one means, roughly, that a value derived using the rules in Figure 6 can also be derived by those in Figure 2. However, this statement needs to be refined, since values in the two semantics are different: possibly infinite objects in the abstract semantics, and capsules in the operational semantics.

We define a relation from capsules to abstract objects, formally express soundness through this relation, and introduce an intermediate semantics to carry out the proof in two steps.

From capsules to infinite objects. Intuitively, given a capsule (v, σ), we get an abstract value by instantiating variables in v with abstract values, in a way consistent with σ. To make this formal, we need some preliminary definitions.

A *substitution* θ is a function from variables to abstract values. We denote by $e\theta$ the abstract expression obtained by applying θ to e. In particular, if e is an open value v, then $\mathrm{v}\,\theta$ is an abstract value. Given an environment σ and a substitution θ, the substitution $\sigma[\theta]$ is defined by:

$$\sigma[\theta](x) = \begin{cases} \sigma(x)\,\theta & x \in \mathit{dom}(\sigma) \\ \theta(x) & x \notin \mathit{dom}(\sigma) \end{cases}$$

Then, a *solution* of σ is a substitution θ such that $\sigma[\theta] = \theta$. Let $\mathsf{Sol}(\sigma)$ be the set of solutions of σ. Finally, if (e, σ) is a capsule, we define the set of abstract expressions it denotes as $[\![e, \sigma]\!] = \{e\theta \mid \theta \in \mathsf{Sol}(\sigma)\}$. Note that $[\![\mathrm{v}, \sigma]\!] \subseteq \mathcal{V}^{\mathrm{a}}$, for any capsule (v, σ). We now show an operational characterization of the semantic equality.

▶ **Theorem 10.** $[\![\mathrm{v}_1, \sigma_1]\!] = [\![\mathrm{v}_2, \sigma_2]\!]$ *iff* $(\mathrm{v}_1, \sigma_1) \approx_\alpha (\mathrm{v}_2, \sigma_2)$*, for some* σ_1, σ_2*-renaming* α*.*

To prove this result we need some auxiliary definitions and lemmas. The *tree expansion* of a capsule (v, σ) is the possibly infinite open value coinductively defined as follows:

$$T(\mathrm{v}, \sigma) = \begin{cases} x & \mathrm{v} = x \text{ and } \mathit{unfold}(x, \sigma) \uparrow \\ \texttt{new}\ C(T(\mathrm{v}_1, \sigma), \ldots, T(\mathrm{v}_n, \sigma)) & \mathit{unfold}(\mathrm{v}, \sigma) = \texttt{new}\ C(\mathrm{v}_1, \ldots, \mathrm{v}_n) \end{cases}$$

The next proposition shows relations between solutions and tree expansion of a capsule.

▶ **Proposition 11.** *Let* (v, σ) *be a capsule and* $\theta \in \mathsf{Sol}(\sigma)$*, then*

1. *if* $\mathit{unfold}(\mathrm{v}, \sigma) \uparrow$ *then* $\mathrm{v} = x$ *and* $x \overset{\sigma}{\leftrightarrow} x$
2. $\mathit{FV}(T(\mathrm{v}, \sigma)) \subseteq \{x \in \mathit{dom}(\sigma) \mid x \overset{\sigma}{\leftrightarrow} x\}$
3. *if* $x \overset{\sigma}{\leftrightarrow} y$ *then* $\theta(x) = \theta(y)$
4. *if* $\mathit{unfold}(\mathrm{v}, \sigma) = \texttt{new}\ C(\mathrm{v}_1, \ldots, \mathrm{v}_n)$ *then* $\mathrm{v}\,\theta = \texttt{new}\ C(\mathrm{v}_1\,\theta, \ldots, \mathrm{v}_n\,\theta)$
5. $\mathrm{v}\,\theta = T(\mathrm{v}, \sigma)\,\theta$

Given a relation α on variables, we will denote by α° the opposite relation and by $=_\alpha$ the equality of possibly infinite open values up-to α, coinductively defined by the following rules:

$$\frac{}{x =_\alpha y}\ x\alpha y \qquad \frac{t_i =_\alpha s_i \quad \forall i \in 1..n}{\texttt{new}\ C(t_1, \ldots, t_n) =_\alpha \texttt{new}\ C(s_1, \ldots, s_n)}$$

It is easy to check that

- α is a σ_1, σ_2-renaming iff α° is a σ_2, σ_1-renaming,
- $(\mathrm{v}_1, \sigma_1) \approx_\alpha (\mathrm{v}_2, \sigma_2)$ iff $(\mathrm{v}_2, \sigma_2) \approx_{\alpha^\circ} (\mathrm{v}_1, \sigma_1)$,
- $t_1 =_\alpha t_2$ iff $t_2 =_{\alpha^\circ} t_1$.

We have the following lemmas:

▶ **Lemma 12.** *$(\mathrm{v}_1,\sigma_1){\approx_\alpha}(\mathrm{v}_2,\sigma_2)$ iff $T(\mathrm{v}_1,\sigma_1){=_\alpha}T(\mathrm{v}_2,\sigma_2)$, for each σ_1,σ_2-renaming α.*

Proof. The proof is immediate by coinduction in both directions. ◀

▶ **Lemma 13.** *If $T(\mathrm{v}_1,\sigma_1){=_\alpha}T(\mathrm{v}_2,\sigma_2)$, where α is a σ_1,σ_2-renaming, then $[\![\mathrm{v}_1,\sigma_1]\!] = [\![\mathrm{v}_2,\sigma_2]\!]$.*

▶ **Proposition 14.** *If $[\![\mathrm{v}_1,\sigma_1]\!] = [\![\mathrm{v}_2,\sigma_2]\!]$ then*
1. *if $\mathit{unfold}(\mathrm{v}_1,\sigma_1)\uparrow$ then $\mathit{unfold}(\mathrm{v}_2,\sigma_2)\uparrow$,*
2. *if $\mathit{unfold}(\mathrm{v}_1,\sigma_1) = \mathtt{new}\ C(\mathrm{v}_{1,1},\ldots,\mathrm{v}_{1,n})$ then $\mathit{unfold}(\mathrm{v}_2,\sigma_2) = \mathtt{new}\ C(\mathrm{v}_{2,1},\ldots,\mathrm{v}_{2,n})$ and, for all $i\in 1..n$, $[\![\mathrm{v}_{1,i},\sigma_1]\!] = [\![\mathrm{v}_{2,i},\sigma_2]\!]$.*

▶ **Lemma 15.** *If $[\![\mathrm{v}_1,\sigma_1]\!]{=}[\![\mathrm{v}_2,\sigma_2]\!]$ then $T(\mathrm{v}_1,\sigma_1) =_\alpha T(\mathrm{v}_2,\sigma_2)$, for some σ_1,σ_2-renaming α.*

Proof of Theorem 10. The right-to-left direction follows from Lemma 12 and Lemma 13, while the other direction follows from Lemma 15 and Lemma 12. ◀

Since by definition $\approx$ is equal to $\approx_\alpha$ for some α, applying Lemma 12 and Lemma 13 we get that if $(\mathrm{v}_1,\sigma_1){\approx}(\mathrm{v}_2,\sigma_2)$ then $[\![\mathrm{v}_1,\sigma_1]\!] = [\![\mathrm{v}_2,\sigma_2]\!]$. Actually we can prove a stronger result:

▶ **Lemma 16.** *If $(\mathrm{v}_1,\sigma_1){\approx_\alpha}(\mathrm{v}_2,\sigma_2)$ for some strict σ_1,σ_2-renaming α, then, for each solution $\theta\in\mathit{Sol}(\sigma_1\cap\sigma_2)$, there are $\theta_1\in\mathit{Sol}(\sigma_1)$ and $\theta_2\in\mathit{Sol}(\sigma_2)$ such that $\mathrm{v}_1\,\theta_1 = \mathrm{v}_2\,\theta_2$ and, for all $x\in\mathit{dom}(\sigma_1\cap\sigma_2)$, $\theta_1(x)=\theta(x)=\theta_2(x)$.*

Soundness statement. We can now formally state the soundness result:

▶ **Theorem 17.** *If $e,\varnothing,\varnothing\Downarrow\mathrm{v},\sigma$, then, for all $v\in[\![\mathrm{v},\sigma]\!]$, $(\mathcal{I}_{\scriptscriptstyle FJ},\mathcal{I}^{co}_{\scriptscriptstyle FJ})\vdash e\Downarrow v$.*

This main result is about the evaluation of source expressions, hence both the environment and the call trace are empty. To carry out the proof we need to generalize the statement.

▶ **Theorem 18** (Soundness). *If $e,\sigma,\varnothing\Downarrow\mathrm{v},\sigma'$, then, for all $\theta\in\mathit{Sol}(\sigma')$, $(\mathcal{I}_{\scriptscriptstyle FJ},\mathcal{I}^{co}_{\scriptscriptstyle FJ})\vdash e\theta\Downarrow\mathrm{v}\,\theta$.*

To show that this is actually a generalization, set $\sigma_1\leq\sigma_2$ if $\mathit{dom}(\sigma_1)\subseteq\mathit{dom}(\sigma_2)$, and, for all $x\in\mathit{dom}(\sigma_1)$, $\sigma_1(x)=\sigma_2(x)$. We use the following lemmas.

▶ **Lemma 19.** *If $\sigma_1\leq\sigma_2$, then $\mathit{Sol}(\sigma_2)\subseteq\mathit{Sol}(\sigma_1)$.*

▶ **Lemma 20.** *If $e,\sigma,\tau\Downarrow\mathrm{v},\sigma'$, then $\sigma\leq\sigma'$.*

In the statement of Theorem 18, thanks to Lemma 20, we know that $\sigma\leq\sigma'$, hence, by Lemma 19, $\theta\in\mathsf{Sol}(\sigma)$, thus $e\theta\in[\![e,\sigma]\!]$. Theorem 18 implies Theorem 17, since, when $\sigma=\varnothing$, e is closed, hence $e\theta=e$, and all elements in $[\![\mathrm{v},\sigma']\!]$ have shape $\mathrm{v}\,\theta$ with $\theta\in\mathsf{Sol}(\sigma')$.

Proof through intermediate semantics. In order to prove Theorem 18, we introduce a new semantics called *intermediate*, defined in Figure 8. Values are those of the abstract semantics, hence calls are of shape $v.m(\overline{v})$ (*abstract* calls). The judgment has shape $e,\rho,S\Downarrow_{\mathrm{IN}}v,S'$, with S,S' sets of abstract calls, ρ map from abstract calls to values. Comparing with $e,\sigma,\tau\Downarrow\mathrm{v},\sigma'$ in the operational semantics, no variables are introduced for calls; ρ and S play the role of the ck and non ck part of τ, respectively, keeping trace of already encountered calls. Moreover, ρ directly associates to a call its value to be used in the checking step, which in σ is associated to the corresponding variable. Finally, S' plays the role of σ', tracing the calls for which the codefinition has been evaluated, hence the checking step will be needed. This correspondence is made precise below. The rules are analogous to those of Figure 6, with the difference that, for an already encountered call $c \in S$, either rule (IN-INVK-OK) or rule (IN-COREC) can be applied. In other words, evaluation of the codefinition is not necessarily triggered when the *first* cycle is detected. This non-determinism makes the relation with the abstract semantics simpler.

By relying on the intermediate semantics, we can prove Theorem 18 by two steps:

1. The operational semantics is sound w.r.t. the intermediate semantics (Theorem 21).
2. The intermediate semantics is sound w.r.t. the abstract semantics (Theorem 23).

At the beginning of Section 4, we mentioned two issues for an operational semantics: representing infinite objects in a finite way, and replacing infinite (regular) proof trees by finite proof trees. This proof technique nicely shows that the two issues are orthogonal: notably, detection of cyclic calls is independent from the format of values.

To express the soundness of the operational semantics w.r.t. the intermediate one, we need to formally relate the two judgments. First of all, a call trace τ is the disjoint union of two maps τ^{ck} and $\tau^{\neg\mathsf{ck}}$ into tagged and non-tagged variables, respectively. Then, given an environment σ, we define the following sets of (operational) calls:

- $S^{\tau} = dom(\tau^{\neg\mathsf{ck}})$
- $S^{\tau,\sigma} = dom(\sigma \circ \tau^{\neg\mathsf{ck}})$, where $\circ$ is the composition of partial functions
- $S^{\tau,\sigma,\sigma'} = S^{\tau,\sigma'} \setminus S^{\tau,\sigma}$

For S set of calls and θ substitution, we abbreviate by S_θ the set of abstract calls $S\,\theta$. Note that $S^{\tau,\sigma}_{\theta} \subseteq S^{\tau}_{\theta}$ and, if $\sigma_1 \leq \sigma_2$, then $S^{\tau,\sigma_1}_{\theta} \subseteq S^{\tau,\sigma_2}_{\theta}$. Finally, $\rho^{\tau}_{\theta}(c\,\theta) = v$ iff $v = \theta(\tau^{\mathsf{ck}}(c))$.

Then, the soundness result can be stated as follows:

▶ **Theorem 21** (Soundness operational w.r.t. intermediate). *If $e, \sigma, \tau \Downarrow \mathrm{v}, \sigma'$ then, for all $\theta \in \mathit{Sol}(\sigma')$, there exists S such that $S^{\tau,\sigma,\sigma'}_{\theta} \subseteq S \subseteq S^{\tau,\sigma'}_{\theta}$ and, $e\theta, \rho^{\tau}_{\theta}, S^{\tau}_{\theta} \Downarrow_{IN} \mathrm{v}\,\theta, S$.*

In particular, the bounds on S ensure that it is empty when $\tau = \emptyset$. Hence, if $e, \sigma, \emptyset \Downarrow \mathrm{v}, \sigma'$ (hypothesis of Theorem 18), then $e\theta, \emptyset, \emptyset \Downarrow_{\mathrm{IN}} \mathrm{v}\,\theta, \emptyset$, that is, the hypothesis of Theorem 23 below holds.

The proof of the theorem uses the following corollary of Lemma 16.

▶ **Corollary 22.** *If $(\mathrm{v}_1, \sigma_1) \approx (\mathrm{v}_2, \sigma_2)$, $\theta_1 \in \mathit{Sol}(\sigma_1)$, $\sigma_1 \leq \sigma_2$, then there is $\theta_2 \in \mathit{Sol}(\sigma_2)$ such that $\mathrm{v}_1\,\theta_1 = \mathrm{v}_2\,\theta_2$ and, for all $x \in dom(\sigma_1)$, $\theta_1(x) = \theta_2(x)$. Moreover, if $\sigma_1 = \sigma_2$, then $\mathrm{v}_1\,\theta_1 = \mathrm{v}_2\,\theta_1$.*

We now state the second step of the proof: the soundness result of the intermediate semantics with respect to the abstract semantics.

▶ **Theorem 23** (Soundness intermediate w.r.t. abstract). *If $e, \emptyset, \emptyset \Downarrow_{IN} v, \emptyset$, then $(\mathcal{I}_{\mathrm{FJ}}, \mathcal{I}^{co}_{FJ}) \vdash e \Downarrow v$.*

The proof uses the bounded coinduction principle (Theorem 1), and requires some lemmas. Recall that $\mathcal{I}_{\mathrm{FJ}} \cup \mathcal{I}^{co}_{\mathrm{FJ}} \vdash e \Downarrow v$ means that the judgment $e \Downarrow v$ has a finite proof tree in the (standard) inference system consisting of FJ rules and coFJ corules.

▶ **Lemma 24.** *If $e, \emptyset, S \Downarrow_{IN} v, S'$ then $\mathcal{I}_{\mathrm{FJ}} \cup \mathcal{I}^{co}_{FJ} \vdash e \Downarrow v$ holds.*

▶ **Lemma 25.** *If $e, \rho, S \cup \{c\} \Downarrow_{IN} v, S'$ holds, and $c \notin S'$, then $e, \rho, S \Downarrow_{IN} v, S'$.*

▶ **Lemma 26.** *If $e, \rho\{c : v'\}, S \Downarrow_{IN} v, S'$ and $c, \rho, S \Downarrow_{IN} v', \emptyset$, then $e, \rho, S \Downarrow_{IN} v, S'$.*

We can now prove Theorem 23.

Proof of Theorem 23. We take as specification the set $A = \{(e, v) \mid e, \emptyset, \emptyset \Downarrow_{\mathrm{IN}} v, \emptyset\}$, and we use bounded coinduction (Theorem 1). We have to prove the following:

$$
\begin{array}{llll}
v \in \mathcal{V}^{\mathrm{a}} & ::=_{\mathrm{CO}} & \texttt{new}\ C(\overline{v}) & \text{possibly infinite object}\\
c & ::= & v.m(\overline{v}) & \text{abstract call}\\
S & ::= & c_1\ \ldots\ c_n\quad (n \geq 0) & \text{set of abstract calls}\\
\rho & ::= & c_1 : v_1 \ldots c_n : v_n\quad (n \geq 0) &
\end{array}
$$

$$
\text{(IN-VAL)}\ \frac{}{v, \rho, S \Downarrow_{\mathrm{IN}} v, \varnothing}
\qquad
\text{(IN-FIELD)}\ \frac{e, \rho, S \Downarrow_{\mathrm{IN}} v, S'}{e.f, \rho, S \Downarrow_{\mathrm{IN}} v_i, S'}
\quad
\begin{array}{l} v = \texttt{new}\ C(v_1, \ldots, v_n)\\ \mathit{fields}(C) = f_1 \ldots f_n\\ f = f_i, i \in 1..n \end{array}
$$

$$
\text{(IN-NEW)}\ \frac{e_i, \rho, S \Downarrow_{\mathrm{IN}} v_i, S_i' \quad \forall i \in 1..n}{\texttt{new}\ C(e_1, \ldots, e_n), \rho, S \Downarrow_{\mathrm{IN}} \texttt{new}\ C(v_1, \ldots, v_n), \bigcup_{i\in 1..n} S_i'}
$$

In all the following rules: $\begin{array}{l} \overline{e} = e_1, \ldots, e_n\\ \overline{v} = v_1 \ldots v_n\\ c = v_0.m(\overline{v})\\ v_0 = \texttt{new}\ C(\_) \end{array}$

$$
\text{(IN-INVK-OK)}\ \frac{\begin{array}{c} e_i, \rho, S \Downarrow_{\mathrm{IN}} v_i, S_i' \quad \forall i \in 0..n\\ e[v_0/\texttt{this}][\overline{v}/\overline{x}], \rho, S \cup \{c\} \Downarrow_{\mathrm{IN}} v, S' \end{array}}{e_0.m(\overline{e}), \rho, S \Downarrow_{\mathrm{IN}} v, \bigcup_{i\in 0..n} S_i' \cup S'}
\quad
\begin{array}{l} c \notin S' \text{ or } c \in S\\ \mathit{mbody}(C, m) = (\overline{x}, e) \end{array}
$$

$$
\text{(IN-INVK-CHECK)}\ \frac{\begin{array}{c} e_i, \rho, S \Downarrow_{\mathrm{IN}} v_i, S_i' \quad \forall i \in 0..n\\ e[v_0/\texttt{this}][\overline{v}/\overline{x}], \rho, S \cup \{c\} \Downarrow_{\mathrm{IN}} v, S'\\ e[v_0/\texttt{this}][\overline{v}/\overline{x}], \rho\{c : v\}, S \Downarrow_{\mathrm{IN}} v, S'' \end{array}}{e_0.m(\overline{e}), \rho, S \Downarrow_{\mathrm{IN}} v, \bigcup_{i\in 0..n} S_i' \cup (S' \setminus \{c\})}
\quad
\begin{array}{l} c \notin S\\ \mathit{mbody}(C, m) = (\overline{x}, e)\\ c \in S' \end{array}
$$

$$
\text{(IN-COREC)}\ \frac{\begin{array}{c} e_i, \rho, S \Downarrow_{\mathrm{IN}} v_i, S_i' \quad \forall i \in 0..n\\ e'[v_0/\texttt{this}][\overline{v}/\overline{x}][u/\texttt{any}], \rho, S \Downarrow_{\mathrm{IN}} v, S' \end{array}}{e_0.m(\overline{e}), \rho, S \Downarrow_{\mathrm{IN}} v, \bigcup_{i\in 0..n} S_i' \cup S' \cup \{c\}}
\quad
\begin{array}{l} c \in S\\ \mathit{co\text{-}mbody}(C, m) = (\overline{x}, e')\\ c \notin \mathit{dom}(\rho) \end{array}
$$

$$
\text{(IN-LOOK-UP)}\ \frac{e_i, \rho, S \Downarrow_{\mathrm{IN}} v_i, S_i' \quad \forall i \in 0..n}{e_0.m(\overline{e}), \rho, S \Downarrow_{\mathrm{IN}} v, \bigcup_{i\in 0..n} S_i'}
\quad \rho(c) = v
$$

Figure 8 coFJ intermediate semantics.

Boundedness For all $(e,v) \in A$, $\mathcal{I}_{\text{FJ}} \cup \mathcal{I}^{co}_{\text{FJ}} \vdash e \Downarrow v$ holds.

Consistency For all $(e,v) \in A$, there exist a rule in the abstract semantics having $e \Downarrow v$ as consequence, and such that all its premises are elements of A.

Boundedness follows immediately from Lemma 24. We now prove consistency.

Consider a pair $(e,v) \in A$, hence we know that $e, \emptyset, \emptyset \Downarrow_{\text{IN}} v, \emptyset$ is derivable. We proceed by case analysis on the last applied rule in the derivation of this judgement.

(IN-val) We know that $e = v = \texttt{new}\ C(v_1, \ldots, v_n)$. We choose as candidate rule (ABS-NEW). We have to show that, for all $i \in 1..n$, $(v_i, v_i) \in A$, that is, $v_i, \emptyset, \emptyset \Downarrow_{\text{IN}} v_i, \emptyset$ holds We can get the thesis thanks to rule (IN-VAL).

(IN-field) We know that $e = e'.f$ and $e', \emptyset, \emptyset \Downarrow_{\text{IN}} \texttt{new}\ C(v_1 \ldots v_n), \emptyset$. We choose as candidate rule (ABS-FIELD), with conclusion $e'.f \Downarrow v_i$. We have to show that $(e', \texttt{new}\ C(v_1 \ldots v_v)) \in A$, that is, $e', \emptyset, \emptyset \Downarrow_{\text{IN}} \texttt{new}\ C(v_1 \ldots v_v), \emptyset$ holds, but this is true by hypothesis.

(IN-new) We know that $e_i, \emptyset, \emptyset \Downarrow_{\text{IN}} v_i, \emptyset$ holds for all $i \in 1..n$. We choose as candidate rule (ABS-NEW). We have to show that, for all $i \in 1..n$, $(e_i, v_i) \in A$, that is, $e_i, \emptyset, \emptyset \Downarrow_{\text{IN}} v_i, \emptyset$ holds, but this is true by hypothesis.

(IN-invk-ok) We know that $e = e_0.m(\overline{e})$, $e_i, \emptyset, \emptyset \Downarrow_{\text{IN}} v_i, \emptyset$ holds for all $i \in 0..n$, $c = v_0.m(\overline{v})$, $\mathit{mbody}(C,m) = (\overline{x}, e')$, and $e'[v_0/\texttt{this}][\overline{v}/\overline{x}], \emptyset, \{c\} \Downarrow_{\text{IN}} v, \emptyset$ holds. We choose as candidate rule (ABS-INVK). We have to show that, for all $i \in 0..n$, $(e_i, v_i) \in A$, and $(e'[v_0/\texttt{this}][\overline{v}/\overline{x}], v) \in A$. That is, that the following judgments hold: $e_i, \emptyset, \emptyset \Downarrow_{\text{IN}} v_i, \emptyset$ for all $i \in 0..n$, and $e'[v_0/\texttt{this}][\overline{v}/\overline{x}], \emptyset, \emptyset \Downarrow_{\text{IN}} v, \emptyset$. The judgments in the first set hold by hypothesis. The last judgment holds thanks to Lemma 25, where $S' = \emptyset$.

(IN-invk-check) We know that $e = e_0.m(\overline{e})$, $e_i, \emptyset, \emptyset \Downarrow_{\text{IN}} v_i, \emptyset$ holds for all $i \in 0..n$, $c = v_0.m(\overline{v})$, $\mathit{mbody}(C,m) = (\overline{x}, e')$, and $e'[v_0/\texttt{this}][\overline{v}/\overline{x}], \{c : v\}, \emptyset \Downarrow_{\text{IN}} v, \emptyset$ holds. We choose as candidate rule (ABS-INVK). We have to show that for all $i \in 0..n$, $(e_i, v_i) \in A$, and $(e'[v_0/\texttt{this}][\overline{v}/\overline{x}], v) \in A$. That is, that the following judgments hold: $e_i, \emptyset, \emptyset \Downarrow_{\text{IN}} v_i, \emptyset$ for all $i \in 0..n$, and $e'[v_0/\texttt{this}][\overline{v}/\overline{x}], \emptyset, \emptyset \Downarrow_{\text{IN}} v, \emptyset$. The judgments in the first set hold by hypothesis. The last judgment holds thanks to Lemma 26, since from the hypothesis we easily get $c, \emptyset, \emptyset \Downarrow_{\text{IN}} v, \emptyset$.

(IN-corec) Empty case since to apply the rule it should be $S \neq \emptyset$.

(IN-look-up) Empty case since to apply the rule it should be $\rho \neq \emptyset$.

◀

7 Related work

As already mentioned, the idea of regular corecursion (keeping track of pending method calls, so to detect cyclic calls), originates from co-SLD resolution [20, 21, 7]. 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 [2, 3], functional [17], and object-oriented [8, 9] paradigm.

Logic paradigm. The above mentioned *co-SLD resolution* [20, 21, 7] is a sound resolution procedure based on cycle detection. That is, the interpreter keeps track of resolved atoms and an atom selected from the current goal can be resolved if it unifies with an atom that has been already resolved. In this way it is possible to define coinductive predicates. Correspondingly, models are subsets of the *complete Herbrand basis*, that is, the set of ground atoms built on arbitrary (finite or infinite) terms, and the declarative semantics is the greatest fixed point of the monotone function associated with a program. Structural resolution [18, 14] (a.k.a. S-resolution) is a proposed generalization for cases when formulas computable at infinity are not regular; infinite derivations that cannot be built in finite time are generated lazily, and only partial answers are shown. More recently, a comprehensive theory has been proposed [11] to provide operational semantics that go beyond loop detection.

Anyway, in coinductive logic programming, only standard coinduction is supported. The notion of `finally` clause, introduced in [2], allows the programmer to specify a fact to be resolved when a cycle is detected, instead of simply accepting the atom. The approach has been refined in [3], following the guidelines given by the formal framework of generalized inference systems. That is, the programmer can write special clauses corresponding to corules, so that, when an atom is found for the second time, standard SLD resolution is triggered in the program enriched by the corules. However, this paradigm is very different from the object-oriented one, since based on relations rather than functions: cycles are detected on the same atom, where input and output are not distinguished, by unification.

Functional paradigm. *CoCaml* (`www.cs.cornell.edu/Projects/CoCaml`) [17, 16] is a fully-fledged extension of OCaml supporting non-well-founded data types and corecursive functions. CoCaml, as OCaml, allows programmers to declare regular values through the let-rec construct, and, moreover, detects cyclic calls as in our approach. However, whereas coFJ immediately evaluates the cyclic call by using the codefinition, the CoCaml approach is in two phases. First, a system of equations is constructed, associating with each call a variable and partially evaluating the body of functions, where calls are replaced with associated variables. Then, the system of equations is given to a *solver* specified in the function definition. Solvers can be either pre-defined or written by the programmer in order to enhance flexibility. An advantage that we see in our approach is that the programmer has to write the codefinition (standard code) rather than working at the meta-level to write a solver, which is in a sense a fragment of the interpreter. A precise comparison is difficult for the lack of a simple operational model of the CoCaml mechanism. In future work, we plan to develop such model, and to relate the two approaches on a formal basis.

Object-oriented paradigm. A previous version of coFJ has been proposed in [8]. At this time, however, the framework of inference systems with corules was still to come, so there was no formal model against which to check the given operational semantics, which, indeed, derived spurious results in some cases, as illustrated in Section 4 at page 16. The operational semantics provided in the current paper solves this problem, and is proved to be sound with respect to the abstract semantics. Moreover, we adopt a simpler representation of cyclic objects through capsules [16]. A type system has been proposed [9] for the previous version of coFJ to prevent *unsafe* use of the "undetermined" value. We leave to further work the investigation of typing issues for the approach presented in this paper.

8 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. Our contribution is foundational: we provide an abstract semantics based on corules and show that it is possible to define a *sound* operational model; such operational semantics is inductive, syntax-directed and deterministic, hence can be directly turned into an interpreter. In order to get a "real-world" language, of course many other issues should be taken into account.

Our prototype implements the *abstract* semantics on top of a Prolog meta-interpreter supporting flexible regular corecursion [3]. In this way, the inference system is naturally translated in Prolog[12], cyclic terms are natively supported, and their equality handled by unification. A fully-fledged interpreter of the *operational* semantics should directly handle these issues and, moreover, attempt at some optimization.

The current paper does not deal with types: an important concern is to guarantee *type soundness*, statically ensuring that an undetermined value never occurs as receiver of field access or method invocation, as investigated in [9] for the previous coFJ version [8].

Another issue is how to train developers to write codefinitions. Standard recursion is non-trivial as well for beginners, whereas it becomes quite natural after understanding its mechanism. For regular corecursion the same holds, with is the additional difficulty of reasoning on infinite structures. Intuitively, the codefinition can be regarded as a base case to be applied when a loop is detected. Moreover, again as for standard recursion, this novel programming style could be integrated with proof techniques to show the correctness of algorithms on cyclic data structures. Such proofs could be mechanized in proof assistants, as Agda, that provide built-in support for coinductive definitions and proofs by coinduction.

Finally, a non-trivial challenge is how to integrate regular corecursion, requiring to detect "the same call", with the notion of mutable state. Likely, some immutability constraints will be needed, or a variant of the model where such a check requires a stateless computation. Another solution is to consider the check as an assertion that can be disabled if the programmer has verified the correctness of the method by hand or assisted by a tool.

The semantics of flexible regular corecursion in the paper is the operational counterpart of that obtained by considering recursive functions as relations, and recursive definitions (with codefinition) as inference systems (with corules). We prove that the operational semantics is *sound* with respect to that interpretation. Obviously, *completeness* does not hold in general, since the abstract semantics deals with not only cyclic data structures (such as $[2, 1]^\omega$), but arbitrary non-well-founded structures (such as the list of natural numbers). Even considering only regular proof trees in the abstract semantics, in some subtle cases there is more than one admissible result[13], whereas the operational semantics, being deterministic, finds "the first" among such results, as reasonable in an implementation . We plan to investigate such completeness issues in further work, also in the more general framework of inference systems, that is, to characterize judgments which have a regular proof tree.

We also plan to study how to deal with flexible corecursion in other programming paradigms, notably in the functional paradigm, and to compare on a formal basis this approach with the CoCaml approach relying on solvers, rather than codefinitions.

As already discussed in the Introduction, lazy evaluation and regular corecursion are complementary approaches to deal with infinite data structures. With the lazy approach, arbitrary (computable) non-well-founded data structures are supported. 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 one_two`, which diverges in Haskell. A natural question is then whether it is possible to extend the regular corecursion approach to manage also non-regular objects, thus overcoming the principal drawback with respect to the lazy approach. A possible interesting direction, exploiting the work of Courcelle [12] on infinite trees, could be to move from regular to *algebraic* objects.

[12] A logic program can be seen as an inference system where judgments are atoms.
[13] For instance, the list with no repetitions extracted from $[1, 2]^\omega$ can be either $[1, 2]$ or $[2, 1]$.

References

1 P. Aczel. An introduction to inductive definitions. In *Handbook of Mathematical logic*. North Holland, 1977.

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

3 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. doi:10.4204/EPTCS.258.1.

4 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, 2017. doi:10.1007/978-3-662-54434-1_2.

5 Davide Ancona, Francesco Dagnino, and Elena Zucca. Reasoning on divergent computations with coaxioms. *PACMPL*, 1(OOPSLA):81:1–81:26, 2017.

6 Davide Ancona, Francesco Dagnino, and Elena Zucca. Modeling infinite behaviour by corules. In *ECOOP'18 - Object-Oriented Programming*, pages 21:1–21:31, 2018.

7 Davide Ancona and Agostino Dovier. A theoretical perspective of coinductive logic programming. *Fundamenta Informaticae*, 140(3-4):221–246, 2015.

8 Davide Ancona and Elena Zucca. Corecursive Featherweight Java. In *FTfJP'12 - Formal Techniques for Java-like Programs*, pages 3–10. ACM Press, 2012.

9 Davide Ancona and Elena Zucca. Safe corecursion in coFJ. In *FTfJP'13 - Formal Techniques for Java-like Programs*, page 2. ACM Press, 2013.

10 Pietro Barbieri, Francesco Dagnino, Elena Zucca, and Davide Ancona. Corecursive Featherweight Java revisited. In Alessandra Cherubini, Nicoletta Sabadini, and Simone Tini, editors, *ICTCS'19 - Italian Conf. on Theoretical Computer Science*, volume 2504 of *CEUR Workshop Proceedings*, pages 158–170. CEUR-WS.org, 2019. URL: http://ceur-ws.org/Vol-2504/paper19.pdf.

11 Henning Basold, Ekaterina Komendantskaya, and Yue Li. Coinduction in uniform: Foundations for corecursive proof search with Horn clauses. In *Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings*, pages 783–813, 2019.

12 B. Courcelle. Fundamental properties of infinite trees. *Theoretical Computer Science*, 25:95–169, 1983.

13 Francesco Dagnino. Coaxioms: flexible coinductive definitions by inference systems. *Logical Methods in Computer Science*, 15(1), 2019. URL: https://lmcs.episciences.org/5277.

14 E.Komendantskaya et al. A productivity checker for logic programming. *Post-proc. LOPSTR'16*, 2017. arXiv:1608.04415.

15 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. doi:10.1145/320384.320395.

16 Jean-Baptiste Jeannin and Dexter Kozen. Computing with capsules. *Journal of Automata, Languages and Combinatorics*, 17(2-4):185–204, 2012. doi:10.25596/jalc-2012-185.

17 Jean-Baptiste Jeannin, Dexter Kozen, and Alexandra Silva. Cocaml: Functional programming with regular coinductive types. *Fundamenta Informaticae*, 150:347–377, 2017.

18 E. Komendantskaya et al. Coalgebraic logic programming: from semantics to implementation. *J. Logic and Computation*, 26(2):745, 2016. doi:10.1093/logcom/exu026.

19 X. Leroy and H. Grall. Coinductive big-step operational semantics. *Information and Computation*, 207(2):284–304, 2009.

20 L. Simon. *Extending logic programming with coinduction*. PhD thesis, University of Texas at Dallas, 2006.

21 L. Simon, A. Bansal, A. Mallya, and G. Gupta. Co-logic programming: Extending logic programming with coinduction. In *ICALP 2007*, pages 472–483, 2007.