Corecursive Featherweight Java *

Davide Ancona
davide@disi.unige.it

Elena Zucca
zucca@disi.unige.it

DISI - Università di Genova
Via Dodecaneso, 35
16146 Genova, Italy

ABSTRACT

Despite cyclic data structures occur often in many application domains, object-oriented programming languages provide poor abstraction mechanisms for dealing with cyclic objects.

Such a deficiency is reflected also in the research on theoretical foundation of object-oriented languages; for instance, Featherweigh Java (FJ), which is one of the most widespread object-oriented calculi, does not allow creation and manipulation of cyclic objects.

We propose an extension to Featherweight Java, called coFJ, where it is possible to define cyclic objects, abstractly corresponding to regular terms, and where an abstraction mechanism, called regular corecursion, is provided for supporting implementation of coinductive operations on cyclic objects.

We formally define the operational semantics of coFJ, and provide a handful of examples showing the expressive power of regular corecursion; such a mechanism promotes a novel programming style particularly well-suited for implementing cyclic data structures, and for supporting coinductive reasoning.

Categories and Subject Descriptors

D.3.1 [**Programming languages**]: Formal Definitions and Theory—*Semantics*; F.3.2 [**Logics and meanings of programs**]: Semantics of Programming Languages—*Operational semantics*; F.3.3 [**Logics and meanings of programs**]: Studies of Program Constructs

General Terms

Languages, Theory

Keywords

coinduction, programming paradigms, regular terms, Java-like languages

*This work has been partially supported by MIUR DISCO - Distribution, Interaction, Specification, Composition for Object Systems.

FTfJP'12, June 12, 2012, Beijing, China

1. INTRODUCTION

In this section we motivate the introduction of regular corecursion by some simple examples, and informally illustrate its operational semantics. All programs defined here manipulate cyclic lists. In Section 2 we give the formal definition of coFJ and in Section 3 we consider more complex cyclic structures and programs. Finally, in Section 4 we outline related and further work.

For the sake of clarity in the examples we use language features not considered in the semantics defined in Section 2; adding such features in the formalized language would be straightforward, without being particularly interesting.

Let us consider the following class declaration:

```
class CycList extends Object {int el; CycList nx;}
```

In FJ such a class is completely useless, since no instance can be created from it: the only available class constructor has two parameters of type **int** and `CycList`; because neither cyclic objects, nor **null** are supported by FJ, there exist no well-typed expressions denoting an instance of `CycList`, and, of course, such a problem is shared by all possible subclasses of `CycList`. To be more precise, there exist well-typed expressions of type `CycList`, but their evaluation never terminates in FJ.

```
class CycListFact extends Object {
    CycList infOcc(int n) {new CycList(n,this.infOcc(n))}
}
```

The expression **new** `CycListFact().infOcc(0)` is well-typed, but its evaluation does not terminate since method `infOcc` attempts to create a list containing infinite occurrences of 0.

There are several solutions to this problem; the one adopted in all mainstream object-oriented languages is the simplest, but also the most primitive one; field `nx` is initialized with a default value (typically **null**), then a finite list is constructed, and, finally, a cycle is introduced by reassigning the proper reference to the field. Such a solution however cannot be adopted by FJ, where assignment is not supported, neither works with **final** fields in mainstream object-oriented languages (for exactly the same reason).

While mainstream object-oriented languages lack any abstraction suitable to manipulate cyclic objects, languages like Haskell [10] exploit the expressive power of lazy evaluation [11] for defining potentially infinite data structures. However, adopting a lazy evaluation strategy for supporting programming with cyclic objects, would be a too radical shift in the semantics of object-oriented languages; indeed, cyclic objects are just a very particular case of infinite objects: abstractly, they correspond to *regular terms* (or trees), that is, finitely branching trees whose depth can be infinite, but that can contain only a finite set of subtrees. For instance, the list made of infinite occurrences of a given number n is regular, whereas the list of all prime numbers is not: the former can be

represented by a cyclic object, the latter can only be implemented indirectly by specific code, with the well-known limitations; for instance, it is possible to check whether a certain predicate holds for all element of a cyclic list, but the same is not feasible for the list of all prime numbers.

What we propose here is a minimal extension to FJ to support cyclic objects and to provide a suitable abstraction for defining methods on them. Such a novel semantics is inspired by the recent results concerning the operational semantics of coinductive Prolog [15, 17, 16] and the implementation of regular corecursion on top of the standard interpreter based on the inductive semantics of the language [2]. The semantic model we consider differs from the conventional FJ semantics in three main aspects:

- objects can be cyclic, hence values can take an equational shape; for instance, $X = \texttt{new}\ C(X)$ is an instance of class C whose unique field contains the object itself.
- methods are *regularly corecursive*: if a recursive call $v.m(\bar{v})$ corresponds to a previous call which is still active on the stack, then such a call terminates immediately, by returning the same value that will be returned by the corresponding call found on the stack.
- a new construct $e.m(\bar{e})$ `with` e' is introduced to provide a *default* value, denoted by e', in place of the value returned by the corecursive call $e.m(\bar{e})$ (see the examples below that motivate its introduction).

  Such a construct does not have a counterpart in corecursive Prolog; due to the asymmetry introduced by the closed world assumption, several predicates (for instance, the predicate `allPos` analogous to the method defined below), do not need a default value; however, the lack of a mechanism to provide a default value in case of corecursion makes the definition of other predicates trickier, as happens for the predicate `member` [2] (the analogous of the method shown later).

Our proposed approach smoothly integrates standard recursion and non cyclic (that is, inductively defined) objects, with corecursion and cyclic (that is, coinductively defined) objects. For simplicity, in the examples that follow, and in the semantics defined in Section 2 we consider only corecursive methods, but in practice, for both performance and semantic reasons, it is possible to adopt a hybrid approach where the user can specify if a method has to exhibit a corecursive behavior or not.

Consider as a first example the following class declarations:

```
class List extends Object { }
class EList extends List { }
class NEList extends List {int el; List nx;}
class CycListFact extends Object {
    NEList infOcc(int n) {
        new NEList(n,this.infOcc(n))}
    NEList zeroOne() {
        new NEList(0,this.oneZero())}
    NEList oneZero() {
        new NEList(1,this.zeroOne())}
}
```

As happens in FJ, one can construct finite lists in the usual way, as in `new NEList(2,new EList())`, but also cyclic ones, as in `infOcc(0)`; such a call would not terminate with the standard semantics, whereas it provides a well-defined value with our proposed semantics: since the recursive call is the same as the initial one, a cyclic object is returned, that is, `L = new NEList(0,L)`.

Similarly, `zeroOne()`, and `oneZero()` return the cyclic lists `L = new NEList(0,new NEList(1,L))` and `L = new NEList(1,new NEList(0,L))`, respectively.

We now consider the more involved method `allPos` which returns true iff all the elements of the list on which it is invoked are positive. The method works correctly for both non cyclic and cyclic lists.

```
class EList extends List {
    bool allPos() { true }
}
class NEList extends List {
    int el; List nx;
    bool allPos() {
        if(this.el <= 0)
            false
        else
            this.nx.allPos() with true}
}
```

If the list is finite, then no regular corecursion is involved, since the same recursive call cannot occur more than once, therefore the default value **true** specified by the **with** clause is never used; if the list is cyclic, but contains a non positive elements, then the method invocation returns the value **false** and corecursion is not applied. The only case where the default value specified by the **with** clause is really needed occurs when the method is invoked on a cyclic list with all positive elements; indeed, without specifying a default value, the result of the method invocation would be the value "undefined", that is, `X = X`.

The pattern used for defining method `member` is similar, but in this case the **with** clause returns **false** which coincides (not by coincidence, in this case) with the value returned by the base case for non cyclic lists.

```
class EList extends List {
    bool member(int i) { false }
}
class NEList extends List {
    int el; List nx;
    bool member(int i) {
        if(this.el == i)
            true
        else
            this.nx.member(i) with false}
}
```

From the examples above, one can be tempted to conclude that the **with** clause is only needed to properly deal with primitive types as **bool** for which recursive equations are not contractive, that is, are not guarded by an instance creation expression, and, hence, the existence of a unique solution is not guaranteed. To show that this is not the case, we define the method `noRep` which, invoked on a possibly infinite (that is, cyclic) list, returns the corresponding finite (non cyclic) list with no repeated elements.

```
class EList extends List {
    EList noRep() { new EList() }
}
class NEList extends List {
    int el; List nx;
    List noRep() {
        let l = this.nx.noRep()
                with new EList() in
            if(l.member(this.el))
                l
            else
                new NEList(this.el,l)}
}
```

For brevity (and efficiency, as well) we have used the **let in** construct, with the standard obvious semantics. Note that, in case `noRep` is invoked on the cyclic list `L = new NEList(0,L)`, the invocation `this.nx.noRep()` would be on exactly the same list, hence, without the **with** clause, the result of `this.nx.noRep()` would be the undefined value, hence `l.member(this.el)` could not be computed (see the formalization in Section 2).

Finally, we end this section with two more examples: the former defines the method `isCyc`, that returns **`true`** iff the list on which it is invoked is cyclic, to show an example where the value returned for the inductive base case is different from the default value of the corresponding **`with`** clause.

```
class EList extends List {
    bool isCyc() { false }
}
class NEList extends List {
    int el; List nx;
    bool isCyc() {
        isCyc(this.nx) with true}
}
```

As a last example, we define a method for removing all negative occurrences from a list. Let us consider the case for non empty lists:

```
    List wrongRemNeg() {
        if(this.el < 0)
            this.nx.wrongRemNeg() with new EList()
        else
            new NEList(this.el,this.nx.wrongRemNeg())}
```

This naive solution fails to behave correctly with cyclic lists containing at least one non negative element.

For instance, `(L = new NEList(1,L)).wrongRemNeg()` returns the non cyclic list **`new`** `NEList(1,new EList())`, instead of the cyclic list itself.

To overcome this problem, `remNeg` for non empty lists can be defined in terms of an auxiliary method taking a boolean parameter `guarded` that indicates whether the returned expression is guarded.

```
class EList extends List {
    EList remNeg() { new EList() }
    EList auxRemNeg(bool guarded) { new EList() }
}
class NEList extends List {
    int el; List nx;
    List remNeg() { this.auxRemNeg(false) }
    List auxRemNeg(bool guarded) {
        if(this.el < 0)
            if(guarded)
                this.nx.auxRemNeg(true)
            else
                this.nx.auxRemNeg(false) with new EList()
        else
            new NEList(this.el,this.nx.auxRemNeg(true))}
```

Initially we assume that the returned expression is not guarded; the condition remains false until a non negative element is encountered. If this happens, then the returned expression is guarded, and, hence, the condition is set permanently to true. If the list contains no non negative elements, and is cyclic, then the condition remains false and the **`with`** clause is finally executed to return the empty list, as expected.

2. FORMAL DEFINITION

The syntax of COFJ is given in Figure 1. We follow usual FJ notations and conventions, notably, we assume infinite sets of *class names* C, *field names* f, *method names* m, and *variables* (*parameter names*) x, and we write $\overline{cd}$ as a shorthand for the sequence $cd_1 \ldots cd_n$, and analogously for other sequences.

There are the following differences w.r.t. FJ: in addition to the basic sets mentioned above, we assume an infinite set of *labels* X, we omit cast expressions for brevity, method invocations have an optional `with` subexpression, and the definition of values is more general.

Indeed, in FJ values and objects (instances of classes) coincide, and have shape $\texttt{new}\ C(\overline{v})$, that is, are (a concrete representation of) inductive terms built by constructor invocations. Here, values

$$
\begin{array}{lcl}
p & ::= & \overline{cd} \\
cd & ::= & \texttt{class}\ C\ \texttt{extends}\ C'\ \{\ \overline{fd}\ \overline{md}\ \} \\
fd & ::= & C\ f\texttt{;} \\
md & ::= & C\ m(\overline{C\ x})\ \{e\texttt{;}\} \\
e & ::= & x \mid e.f \mid e.m(\overline{e})\ [\texttt{with}\ e'] \mid \texttt{new}\ C(\overline{e}) \\
 & & \\
u, v & ::= & \texttt{new}\ C(\overline{v}) \mid X = v \mid X
\end{array}
$$

Figure 1: COFJ syntax

are generalized, that is, they can be annotated with labels, and a (sub)value can be a (reference to a) label, expected to annotate an enclosing value. Objects are values which are not labels, that is, of form $X_1 = \ldots\ X_n = \texttt{new}\ C(\overline{v})$, abbreviated $\overline{X =}\ \texttt{new}\ C(\overline{v})$ with our convention. Values annotated with more than one label, like, e.g., `X = Y = new C(X)`, are obtained by reduction, see Figure 3.

We expect the result of evaluating a top-level expression to be *closed*, that is, with all references bound to existing labels. Note that there are values which are *not* expressions, since, to keep the language minimal, a cyclic object can only be obtained as result of a method invocation, as shown in the examples of previous section.

Closed values are a concrete representation of regular terms built by constructor invocations and "undefined". Note that the only closed values which are not objects have shape $X_1 = \ldots\ X_n = X_i$, with $i \in 1..n$, and are representations of "undefined". All the closed values representing the same regular term, as, for instance, the following:

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

are considered implicitly equal, and an analogous assumption holds for open values as well. We expect that, if the results of two expressions e and e' are the same modulo this equality, then e and e' can replace each other in any context. For lack of space, we omit the standard formal definition of value equality and the formal statement and proof of this congruence result..

Note that values which are not objects (notably, representations of "undefined" and labels) cannot be safely used as receivers in field accesses and method invocations, but can be passed as arguments and obtained as result of field access and method invocation.

The big-step semantics $e, \sigma \Downarrow v$ returns the result v, if any, of evaluating an expression e in the context of an *environment* σ keeping track of pending method invocations. Formally, σ is a map from expressions of the form $v.m(\overline{v})$, which we call *(invocation) redexes*, to labels X. We prefer a big-step style since small-step semantics would require to explicitly handle stacks of environments, and we are not considering (yet) the issue of detecting stuck expressions in this paper (see the Conclusion).

The rules are given in Figure 2. For lack of space, we omit technical details which are exactly as in FJ, notably, the formal definitions of parallel substitution and auxiliary functions *fields* and *mbody*.

Rule (FIELD) models field access. The receiver expression is evaluated, and its result is expected to be an object. The standard FJ function *fields* retrieves the sequence of the field names of its class, and, if the selected field is actually a field of the class, the corresponding value is returned as result. Note that this value could contain references to the enclosing receiver object, which must be unfolded.

For instance, given the class

```
class C {
```

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

$$\text{(INVK)}\ \frac{e,\sigma \Downarrow v \quad \overline{e},\sigma \Downarrow \overline{v} \quad e[v/\texttt{this}][\overline{v}/\overline{x}],\sigma \cup \{v.m(\overline{v}) \mapsto X\} \Downarrow u}{e.m(\overline{e})\ [\texttt{with}\ \_],\sigma \Downarrow X = u}\quad \begin{array}{l} v = \overline{X=}\,\texttt{new}\ C(\_) \\ mbody(C,m) = (\overline{x},e) \\ v.m(\overline{v}) \notin dom(\sigma) \\ X \text{ fresh} \end{array}$$

$$\text{(COREC)}\ \frac{e,\sigma \Downarrow v \quad \overline{e},\sigma \Downarrow \overline{v}}{e.m(\overline{e}),\sigma \Downarrow X}\quad \sigma(v.m(\overline{v})) = X$$

$$\text{(WITH)}\ \frac{e,\sigma \Downarrow v \quad \overline{e},\sigma \Downarrow \overline{v} \quad e',\sigma \Downarrow u}{e.m(\overline{e})\ \texttt{with}\ e',\sigma \Downarrow u}\quad \sigma(v.m(\overline{v})) = X$$

$$\text{(NEW)}\ \frac{\overline{e},\sigma \Downarrow \overline{v}}{\texttt{new}\ C(\overline{e}),\sigma \Downarrow \texttt{new}\ C(\overline{v})}\quad fields(C) = \overline{f}$$

Figure 2: coFJ big-step rules

```
  C f;
}
```

the field access v.f, with

$$v = \texttt{X=new C(Y=new C(X))},$$

is reduced to

$$u = \texttt{Y=new C(X=new C(Y=new C(X)))},$$

since $fields(\texttt{C}) = \texttt{f}$ and $(\texttt{Y=new C(X)})[v/\texttt{X}] = u$.

There are three rules which model method invocation. In all of them, the receiver and argument expressions are evaluated first, obtaining an invocation redex $v.m(\overline{v})$. Then, the behaviour is different depending whether this redex has been already encountered in the current stack of method invocations (formally, is defined in σ).

If this is not the case, then the method invocation is handled as usual, see rule (INVK), that is, the result of the receiver expression is expected to be an object, and method look-up is performed, starting from its class, by the standard function $mbody$, getting the corresponding method parameters and body. Then, the result of the invocation is obtained by evaluating the method body where the receiver object replaces this and the arguments replace the parameters. However, there is a difference w.r.t. this standard FJ semantics, that is, evaluation of the method body is performed keeping track of the redex just encountered (formally, adding in σ an association from this redex to a fresh label X). In this way, method invocations leading to the same redex will be no longer handled by this rule (hence avoiding non termination). Moreover, when the evaluation of the method body is completed, references to X in the resulting value (due to recursive method invocations leading to the same redex handled by rule (COREC)) are bound. This mechanism makes it possible to obtain a cyclic object as the result of a method invocation.

Rules (COREC) and (WITH) handle the case of an invocation redex which has been already encountered. In the former, no **with** expression is provided in the method invocation, hence the result is a reference to the label of the previous occurrence. In the latter, a **with** expression is provided, hence is evaluated.

For instance, given the classes

```
class C {
  Object f;
}
class A {
  C m1() {this.m2()); }
  C m2() {new C(this.m1()); }
}
```

the method invocation **new** A().m1() is reduced to the cyclic object X=Y=**new** C(X) (equivalent to X=**new** C(X)) as shown in Figure 3. If method m2 were, instead,

```
  Object m2() {
    new C(this.m1() with new A()); }
```

then the instantiation of rule (COREC) would be replaced by an instantiation of rule (WITH), and the method invocation **new** A().m1() would be reduced to the non cyclic object X=Y=**new** C(**new** A()) (equivalent to **new** C(**new** A())).

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

We show the consistency of the calculus by the following two theorems. The former states that the evaluation of an expression in a given environment returns, if any, a value whose free labels are used in the environment (hence in particular the evaluation in the empty environment returns a closed value). The latter states that coFJ semantics extends the standard recursive semantics, that is, if we get a result by FJ semantics, then we get the same result by the corecursive semantics. Of course the converse does not hold, since corecursive semantics can return a value in cases where recursive semantics does not terminate.

Let us denote by $FL(v)$ the set of free labels in value v, defined in the obvious way, and by $img(\sigma)$ the image of σ.

THEOREM 2.1. *If $e,\sigma \Downarrow v$, then $FL(v) \subseteq img(\sigma)$.*

PROOF. By induction on the rules defining $e,\sigma \Downarrow v$.

- (FIELD) By inductive hypothesis $FL(v) \subseteq img(\sigma)$, hence $FL(v_i[v/\overline{X}]) \subseteq img(\sigma)$.
- (INVK) By inductive hypothesis $FL(u) \subseteq img(\sigma) \cup \{X\}$, hence $FL(X = u) \subseteq img(\sigma)$.
- (COREC) Trivially by the side condition.
- (WITH) and (NEW): trivially by inductive hypothesis. □

The standard syntax and recursive semantics of FJ in big-step style are reported in Figure 4.

THEOREM 2.2. *For e expression and v value in* FJ*, if $e \Downarrow_{\text{FJ}} v$, then $e,\sigma \Downarrow v$ for all σ.*

PROOF. By induction on the rules defining $e \Downarrow_{\text{FJ}} v$.

- (FJ-FIELD) The premise of rule (FIELD) holds by inductive hypothesis, hence the consequence as well, where, since $\overline{X}$ is the empty sequence, $v_i[v/\overline{X}] = v_i$.
- (FJ-INVK) By inductive hypothesis the premises of rule (INVK) hold, hence the consequence as well, where, since u is an FJ value, hence does not contain labels, $X = u$ is equivalent to u.
- (FJ-NEW) is exactly analogous to (NEW) in coFJ, hence the thesis trivially holds. □

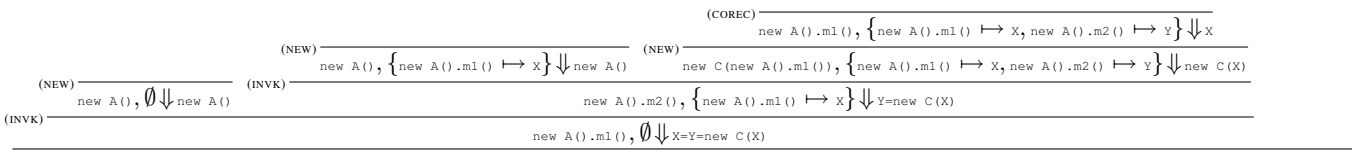

Figure 3: Example of reduction

$$
\begin{array}{lll}
e & ::= & x \mid e.f \mid e.m(\overline{e}) \mid \texttt{new}\ C(\overline{e}) \\
u, v & ::= & \texttt{new}\ C(\overline{v})
\end{array}
$$

$$
\text{(FJ-FIELD)}\ \frac{e \Downarrow_{\text{FJ}} v}{e.f \Downarrow_{\text{FJ}} v_i}\quad \begin{array}{l} v = \texttt{new}\ C(\overline{v}) \\ \mathit{fields}(C) = \overline{f} \\ f = f_i, i \in 1..n \end{array}
$$

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

$$
\text{(FJ-NEW)}\ \frac{\overline{e} \Downarrow_{\text{FJ}} \overline{v}}{\texttt{new}\ C(\overline{e}) \Downarrow_{\text{FJ}} \texttt{new}\ C(\overline{v})}\quad \mathit{fields}(C) = \overline{f}
$$

Figure 4: FJ syntax and big-step rules

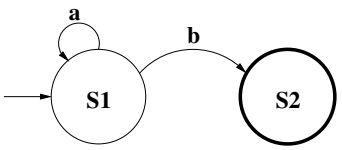

Figure 5: A deterministic finite automaton recognizing the language `a*b`

3. PROGRAMMING WITH COFJ

In this section we present some more significant examples of COFJ programming, that show the usefulness of regular terms and corecursion.

3.1 Finite automata and regular languages

We now consider a classical application from formal languages, by defining a method that succeeds if and only if the language generated by an extended right linear grammar is included in the language recognized by a finite deterministic automaton. Cyclic objects can be exploited for representing automata and regular grammars.

An automaton is represented by its unique initial state.

```
class State extends Object {
    bool isFinal; AdjList trans;
}
class AdjList extends Object { }
class EAdjList extends AdjList{ }
class NEAdjList extends AdjList{
    char sym; State st; AdjList nx;
}
```

Let us consider the automaton depicted in Figure 5, where S1 (pointed by the straight arrow in the picture) is the initial state, and S2 (with a thicker circle) is final. Such an automaton can be represented by the following instance of `State`:

```
S = new State(false,
     new NEAdjList('a',S,
      new NEAdjList('b',new State(true,new EAdjList()),
                        new EAdjList()))))
```

The instance variable `isFinal` indicates whether a state is final or not, whereas `trans` corresponds to the list of possible transitions, that is, the adjacency list of the current state: each item of the list consists of a symbol (represented by **`char`**) and of the corresponding target state. Class `AdjList` represents adjacency lists, where the subclasses `EAdjList` and `NEAdjList` represent empty and non empty lists, respectively.

We recall that an extended right linear grammar is a grammar where all productions have shape either $N_1 ::= a$, or $N_1 ::= \epsilon$, or $N_1 ::= wN_2$, where N_1 and N_2 are two non-terminal symbols, a is a terminal symbol, w is a (possibly empty) string of terminal symbols, and ϵ is the empty string.

A grammar is represented by its main non-terminal symbol.

```
class NonTermDef extends Object { }
class EmptyString extends NonTermDef { }
class Concat extends NonTermDef{
    char sym; NonTermDef nx; }
class Union extends NonTermDef {
    NonTermDef nt1; NonTermDef nt2; }
```

The encoding of the definition of a non-terminal symbol (that is, all its productions) is based on the conventional mapping to set expressions built on top of the singleton set containing the empty string (`EmptyString`), and the concatenation (`Concat`) and union (`Union`) operator. For instance, let us consider the following right linear grammar:

```
A ::= b | aA
```

Such a grammar can be encoded by the following cyclic object:

```
N = new Union(new Concat('b',new EmptyString()),
     new Concat('a',N))
```

Given the encoding of automata and grammars as described above, it is possible to define the method `included` for `NonTermDef` objects that takes as parameter an automaton (that is, an instance of class `State`), and returns **`true`** iff the language generated by the grammar is included in the language recognized by the automaton.

```
class EmptyString extends NonTermDef {
    bool included(State s) { s.isFinal }}
class Concat extends NonTermDef {
    char sym; NonTermDef nx;
    bool included(State s) {
        let ns = s.trans.getState(this.sym)
        in if (ns == null) false
           else this.nx.included(ns)
                with true
    }}
class Union extends NonTermDef {
    NonTermDef nt1; NonTermDef nt2;
    bool included(State s) {
        this.nt1.included(s) with true
        &&
        this.nt2.included(s) with true
    }}
```

The case for the empty string is straightforward: the empty string is accepted only if the initial state of the automaton is also final.

For concatenation, the auxiliary method `getState` (whose definition has been omitted) is employed: it is invoked on an adjacency

list with an argument `sym` of type **`char`**, to find an outgoing edge labeled with `sym`; if found, the corresponding target state is returned, otherwise **`null`** is returned.[1]

The case for union is simple: the union of two languages is contained in the automaton iff both are contained in it.

For all corecursive invocations (both in `Concat` and `Union`) the default value **`true`** is specified by the **`with`** clause. This corresponds to the intuition that if an active invocation of `included` is encountered again, then a cyclic path in the automaton has been detected corresponding to the acceptance of the language.

The careful reader will notice that the definition of `included` is not completely correct, since it fails to correctly deal with grammars that generate the empty set. Consider for instance the grammar $A ::= aA$: its generated language is the empty set, that is recognized by any automaton; however, method `included` in `Concat` does not succeed if there are no outgoing edges labeled with a (in other words the presented solution works correctly when grammars are interpreted coinductively, rather than inductively). To overcome this problem, we introduce the method `emptySet` that checks whether a grammar generates the empty set; then we extend methods `included`, to first check whether the grammar corresponding to the object `this` generates the empty set; if so, **`true`** is returned.

```
class EmptyString extends NonTermDef {
    bool included(State s) { this.emptySet() || ... }
    bool emptySet() { false }
}
class Concat extends NonTermDef {
    char sym; NonTermDef nx;
    bool included(State s) { this.emptySet() || ... }
    bool emptySet() {
        this.nx.emptySet() with true;
    }
}
class Union extends NonTermDef {
    NonTermDef nt1; NonTermDef nt2;
    bool included(State s) { this.emptySet() || ... }
    bool emptySet() {
        this.nt1.emptySet() with true
        &&
        this.nt2.emptySet() with true
    }
}
```

3.2 Repeating decimals

It is well-known that every rational number can be represented by a repeating decimal, that is, a cyclic lists of digits. For simplicity we only consider the interval $[0, 1]$, although the code presented below can be easily extended to work with the whole set of rational numbers.

```
class RepDec extends Object { int dig; RepDec nx; }
```

As an example, the object

```
N = new RepDec(5,P = new RepDec(7,new RepDec(2,P)))
```

corresponds to the repeating decimal `N` $= 0.5\overline{72}$. In terms of fractions, `N` equals $\frac{63}{110}$. Indeed, $10 * N = 5 + 0.\overline{72}$, and $100 * 0.\overline{72} = 72 + 0.\overline{72}$ (multiplying a repeating decimal by 10^e, with $e > 0$, is equivalent to a left shift of e positions). The above gives rise to the following equations: $10N$=5+P,$100P$=72+P. Therefore `P`$= \frac{8}{11}$, and N=$\frac{5}{10} + \frac{4}{55} = \frac{55+8}{110} = \frac{63}{110}$. A terminating decimal can be uniformly represented by a repeating decimal as well; for instance, 0.3 is represented by the object

```
D = new RepDec(3,Z = new RepDec(0,Z))
```

[1]The **`null`** reference has been introduced just to make the example code more compact.

We now define a method to compute the addition between two repeating decimals `d1` and `d2`. Since the operands have infinite digits, we cannot simply mimic the conventional algorithm for addition, because the notion of least significant digit does not make sense in this case. We first define the following two auxiliary methods that compute the repeating decimal corresponding to the results and carries, respectively, of digit-wise addition of the digits of `d1` and `d2`.

```
class RepDec extends Object {
 int dig; RepDec nx;
 RepDec res(RepDec d) {
    new RepDec((this.dig+d.dig)%10,this.nx.res(d.nx))}
 RepDec carry(RepDec d) {
    new RepDec((this.dig+d.dig)/10,this.nx.carry(d.nx))}
}
```

The two methods `res` and `carry` take an argument `d`, compute the addition and the carry, respectively, for the two most significant digits of `this` and `d`, and then continue corecursively for the rest of the digits of `this` and `d`. No **`with`** clause is required, because the recursive equations corresponding to the result are always contractive.

The method `add` is defined in terms of `res` and `carry`.

```
class Pair extends Object { RepDec res; int carry; }
class RepDec extends Object {
    int dig; RepDec nx;
    bool isZero() {
        if(this.dig > 0)
            false
        else
            this.nx.isZero() with true}
    Pair add(RepDec d) {
        if(this.isZero())
            new Pair(d,0)
        else
            let r = this.res(d) in
            let c = this.carry(d) in
            let p = r.add(c.nx) in
            new Pair(p.res,p.carry+c.dig)}
}
```

The auxiliary method `isZero` checks whether `this` is the repeating decimal corresponding to 0.

Method `add` returns a pair, where the first component is the repeating decimal corresponding to the result, and the second is the carry for the next more significant position. If `this` is 0, then the result is `d`, and the carry is 0. Otherwise, the auxiliary methods `res` and `carry` are invoked; then the carry digits have to be considered: first a left shift of one position is required (this is obtained by simply selecting field `nx`); then the shifted carry is corecursively added to `r`; indeed, the carry digit generated at position i (corresponding to the power 10^{-i}) must be added to the digit of `r` at position $i - 1$. Finally, the carry `p.carry` obtained by such an addition has to be added to the most significant digit `c.dig` of `c`, to properly compute the carry digit for the next more significant position.

The computation terminates because of cyclicity, and because each position can yield a carry of 1 just once.

We finally recall that repeating decimals provide no unique representation for some rational numbers: for instance $0.4\overline{9}$ equals 0.5; however, method `add` works correctly independently of the representation of operands. A practical way for defining the equality test is to implement it in terms of subtraction (that can be defined in a very similar way as addition); however, one may also consider a normalization procedure that, for instance, prefers 0.5 over $0.4\overline{9}$.

3.3 Graphs

Last but not least, we end this section by showing how a classical graph algorithm can be concisely implemented with regular corecursion. Graphs are perhaps the most interesting application domain of regular corecursion: they are the prototypical example

of cyclic structure, and arise in so many important areas of computer science.

The depth-first search algorithm, which is at the basis of several other graph algorithms, can be conveniently implemented with regular corecursion. The following example shows the implementation of method `connectedTo` for testing connectivity of a (either directed or undirected) graph.

In a similar way as that shown for automata, a graph can be represented by one of its vertices, together with its adjacency list.

```
class Vertex extends Object {
    int id; AdjList adjVerts;
}
class AdjList extends Object { }
class EAdjList extends AdjList{ }
class NEAdjList extends AdjList{
    Vertex vert; AdjList next;
}
```

Every vertex is represented by its `id` (assumed to be unique) and its list `adjVerts` of adjacent vertices.

The method invocation `v.isConnected(id)` returns true if and only if there exists a (possibly empty) path from `v` to the vertex identified by `id`. The method is defined for both vertices and adjacency lists.

```
class Vertex extends Object {
    int nodeId; AdjList adjVerts;
    bool isConnected(int id) {
        this.id == id || this.adjVerts.isConnected()
    }
}
class EAdjList extends AdjList{
    bool isConnected(int id) { false }
}
class NEAdjList extends AdjList{
    Vertex vert; AdjList next;
    bool isConnected(int id) {
        this.vert.isConnected(id) with false
        ||
        this.next.isConnected(id)
    }
}
```

While the definition of `isConnected` for the empty adjacency list is obvious, the remaining cases deserve some explanation.

In `Vertex` method `isConnected` checks whether the identity of the current vertex (bound to `this`) equals the identity of the searched vertex, or whether there exists a path connecting the two vertices that contains one of the adjacent vertices of `this`.

In `NEAdjList` method `isConnected` has to check that there exists a path connecting one of the vertices in the adjacency list with the vertex specified by `id`. Since the algorithm implicitly assumes that adjacency lists are not cyclic, the only invocation that can detect a cycle is `this.vert.isConnected(id)`; therefore, this is also the only invocation where the **`with`** clause is required. If a cycle is encountered, then the corresponding path is assumed not to contain the vertex specified by `id`, therefore **`false`** is returned by the method invocation.

A short-circuit evaluation for the or operator is not required for the correctness of the implementation, even though it makes it more efficient.

4. RELATED WORK AND CONCLUSION

As already mentioned, this paper is inspired by recent work on coinductive logic programming and regular recursion in Prolog. Simon et al. [15, 17, 16] have proposed coinductive SLD resolution (abbreviated by coSLD) as an operational semantics for logic programs interpreted coinductively: the coinductive Herbrand model is the greatest fixed-point of the one-step inference operator. This can be proved equivalent to the set of all ground atoms for which there exists either a finite or an infinite SLD derivation [17]. Coinductive logic programming has proved to be useful for formal verification [12, 13], static analysis and symbolic evaluation of programs [5, 4, 6].

Regular corecursion in Prolog has been investigated by one of the authors of this paper as a useful abstraction for programming with cyclic data structures. To our knowledge, no similar approaches have been considered for functional programming; although the problem has been already considered [19, 9], the proposed solutions are based on the use of specific and complex datatypes, but no new programming abstraction is proposed.

A related stream of work is that on initialization of circular data structures [18, 8, 14].

The calculus presented in this paper introduces a novel programming style, which smoothly incorporates support for cyclic data structures and coinductive reasoning in the object-oriented ; such a style could be conveniently integrated with proof assistants, as Coq [7], that provide built-in support for coinductive definitions and proofs by coinduction, to formally proof the correctness of algorithms on cyclic data structures.

In comparison with the more foundational studies [1, 3] on the use of coinductive big-step operational semantics of Java-like languages for proving type soundness properties, the main contribution and aim of this paper is to propose a novel programming paradigm to support useful abstractions for the convenient creation and manipulation of cyclic data structures.

The current work is intended as a first step, devoted to present the novelties of the approach, and there is plenty of interesting directions for further research. The most obvious question is how to guarantee type soundness. The standard FJ type system should be enriched to avoid stuck reductions due to non-object values (that is, "undefined" or labels) in receiver position, likely in a similar way to techniques used to avoid **`null`** in receiver position. On the foundational side, we plan to explore the relation between our operational semantics and the more abstract semantics of corecursive definitions as greatest fixed points. On the more practical side, we also plan to investigate implementation techniques and the portability of this new programming paradigm to mainstream languages.

Acknowledgements We warmly thank the anonymous referees for their helpful comments.

5. REFERENCES

[1] D. Ancona. Coinductive big-step operational semantics for type soundness of Java-like languages. In *FTfJP '11*, pages 5:1–5:6. ACM, 2011.

[2] D. Ancona. Regular corecursion in Prolog. In *SAC 2012*, pages 1897–1902, 2012.

[3] D. Ancona. Soundness of object-oriented languages with coinductive big-step semantics. In *ECOOP 2012*, 2012. To appear.

[4] D. Ancona, A. Corradi, G. Lagorio, and F. Damiani. Abstract compilation of object-oriented languages into coinductive CLP(X): can type inference meet verification? In *FoVeOOS 2010, Revised Selected Papers*, volume 6528 of *LNCS*, 2011.

[5] D. Ancona and G. Lagorio. Coinductive type systems for object-oriented languages. In *ECOOP 2009*, volume 5653 of *LNCS*, pages 2–26, 2009.

[6] D. Ancona and G. Lagorio. Idealized coinductive type systems for imperative object-oriented programs. *RAIRO - Theoretical Informatics and Applications*, 45(1):3–33, 2011.

[7] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive*

Constructions. Texts in Theoretical Computer Science. Springer Verlag, 2004.

[8] M. Fähndrich and S. Xia. Establishing object invariants with delayed types. In *OOPSLA 2007*, pages 337–350. ACM Press, 2007.

[9] N. Ghani, M. Hamana, T. Uustalu, and V. Vene. Representing cyclic structures as nested datatypes. In *TFP*, pages 173–188, 2006.

[10] P. H., J. H., S. L. Peyton Jones, and P. Wadler. A history of haskell: being lazy with class. In *History of Programming Languages Conference (HOPL-III)*, pages 1–55, 2007.

[11] John Launchbury. A natural semantics for lazy evaluation. In *POPL*, pages 144–154, 1993.

[12] R. Min and G. Gupta. Coinductive logic programming and its application to boolean sat. In *FLAIRS Conference*, 2009.

[13] N.Saeedloei and G. Gupta. Verifying complex continuous real-time systems with coinductive CLP(R). In *LATA 2010*, LNCS. Springer, 2010.

[14] Xin Qi and Andrew C. Myers. Masked types for sound object initialization. In Z. Shao and B. C. Pierce, editors, *POPL 2009*, pages 53–65. ACM Press, 2009.

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

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

[17] L. Simon, A. Mallya, A. Bansal, and G. Gupta. Coinductive logic programming. In *ICLP 2006*, pages 330–345, 2006.

[18] A. J. Summers and P. Müller. Freedom before commitment - a lightweight type system for object initialisation. In *OOPSLA 2011*. ACM Press, 2011.

[19] F. A. Turbak and J. B. Wells. Cycle therapy: A prescription for fold and unfold on regular trees. In *PPDP*, pages 137–149, 2001.