Safe Corecursion in coFJ

Davide Ancona
davide.ancona@unige.it

Elena Zucca
elena.zucca@unige.it

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

ABSTRACT

In previous work we have presented COFJ, an extension to Featherweight Java that promotes *coinductive programming*, a sub-paradigm expressly devised to ease high-level programming and reasoning with cyclic data structures.

The COFJ language supports cyclic objects and *regularly corecursive* methods, that is, methods whose invocation terminates not only when the corresponding call trace is finite (as happens with ordinary recursion), but also when such a trace is infinite but cyclic, that is, can be specified by a regular term, or, equivalently, by a finite set of recursive syntactic equations.

In COFJ it is not easy to ensure that the invocation of a corecursive method will return a well-defined value, since the recursive equations corresponding to the regular trace of the recursive calls may not admit a (unique) solution; in such cases we say that the value returned by the method call is *undetermined*.

In this paper we propose two new contributions. First, we design a simpler construct for defining corecursive methods and, correspondingly, provide a more intuitive operational semantics. For this COFJ variant, we are able to define a type system that allows the user to specify that certain corecursive methods cannot return an undetermined value; in this way, it is possible to prevent *unsafe* use of such a value.

The operational semantics and the type system of COFJ are fully formalized, and the soundness of the type system is proved.

1. INTRODUCTION

In previous work we have presented COFJ [7], an extension to Featherweight Java that promotes *coinductive programming*, a sub-paradigm expressly devised to ease high-level programming and reasoning with cyclic data structures.

In COFJ objects are purely functional like in FJ, but differently from FJ, it is possible to define cyclic objects and methods are *regularly corecursive*: a method invocation terminates not only when the corresponding call trace is finite (as happens with ordinary recursion), but also when such a trace is infinite but cyclic, that is, can be specified by a regular term, or, equivalently, by a finite set of recursive syntactic equations; similarly, the returned value of a method invocation may correspond to a solution of a finite set of recursive syntactic equations, therefore it may be a cyclic object.

Let us consider, as a first simple example, the following COFJ classes.

```
class RepDecFact extends Object {
   RepDec zero() { // returns the repeating decimal 0
      new RepDec(0,zero()) with res
   }
}
class RepDec extends Object {
    int digit;
    RepDec next;
    RepDec comp() { // returns 1 - this
        new RepDec(9-this.digit,this.next.comp())
        with res
    }
    bool isZero() { // check if this is zero
       (digit==0 && this.next.isZero()) with true
    }
}
```

Class `RepDec` implements the closed interval $[0, 1]$ of rational numbers with cyclic sequences of digits (that is, repeating decimals); for instance, $\frac{7}{45} = \frac{1}{10} + \frac{1}{18}$ is represented by the cyclic sequence $15555\ldots$.

In COFJ each class is equipped with a unique implicitly declared constructor which takes as arguments initialization values for all inherited and declared fields, in the order they are inherited and declared; for intance, the constructor for `RepDec` has two parameters of type **`int`** and `RepDec`, respectively.

While in FJ it would not be possible to create an instance of `RepDec`, since field `next` has type `RepDec`, method `zero()` in factory class `RepDecFact` returns the cyclic sequence of 0, thanks to regular corecursion. Indeed, if o=**`new`** `RepDecFact()`, then the evaluation of the expression o`.zero()` yields the cyclic trace of invocations o`.zero()`o`.zero()`..., hence it terminates; furthermore, the sub-expression **`with res`** specifies that the value returned by the method is the solution of the recursive equation res=**`new`** `RepDec(0,`res`)` generated by the cyclic trace. More precisely, as specified by the operational semantics presented in Section 3, **`res`** is a special variable (like **`this`**) which denotes a well-defined value only in case the generated recursive equations admit a unique solution; otherwise, **`res`** denotes an *undetermined* value on which field selection and method invocation are undefined (hence, the evaluation gets stuck).

The body of a corecursive method consists of a **with** expression where its rhs is evaluated only when a cycle in the call trace is detected (that is, the method terminates corecursively).

Similarly, the expression e`.comp()` returns the cyclic sequence of digits corresponding to the complement of e, that is, $1 - r$, if e denotes the rational r. For instance, let e denotes the sequence $15555\ldots$; then the result of $e.comp()$ is the solution (projected to nxt) of the following equations:

nxt = **`new`** `RepDec(8,`res`)`
res = **`new`** `RepDec(4,`res`)`

If we consider method `isZero`, we notice that the rhs of **with** is the literal **`true`**, rather then the variable **`res`**, because in this case the system of equations associated with a cyclic call trace may have more solutions. Indeed, if `e`=**`new`** `RepDecFact().zero()`, then e`.isZero()` yields the cyclic trace e`.isZero()` e`.isZero()` ..., and the corresponding returned value is the solution of the recur-

sive equation `res = (0==0) && res` for which `res = false` and `res = true` are both valid solutions, hence the value associated with *res* is *undetermined*. In general, the existence of a unique solution, and the ability of computing it, is guaranteed only when the recursive equations are guarded by object constructors. When equations are not guarded, the programmer has to specify a value different from **res** in the rhs of **with**; for instance, for method `isZero` the wanted behavior is obtained by returning the literal **true**.

In this paper we propose two new contributions. First, we design a simpler construct for defining corecursive methods and, correspondingly, provide a more intuitive operational semantics. For this COFJ variant, we are able to define a type system that allows the user to specify that certain corecursive methods cannot return an undetermined value; in this way, it is possible to prevent *unsafe* use of such a value.

In Section 2 we informally illustrate COFJ and the operational semantics of controlled regular corecursion. In Section 3 we give the formal definition of COFJ, in Section 4 the type system and the related soundness result. Finally, in Section 5 we outline related and further work.

The Appendix includes the formal definition of FJ, for reference, and additional, more complex, examples.

In [8], we have provided a derived semantics of COFJ by translation into coinductive logic programming. A prototype implementing this translation can be found at `http://www.disi.unige.it/person/AnconaD/Software/CoFJ`.

2. EXAMPLES

This section is a gentle introduction to COFJ; all programs defined here manipulate cyclic lists, examples on more advanced cyclic structures can be found in Appendix 2. For the sake of clarity in the examples we use language features not considered in the calculus defined in Section 3 and 4; adding such features would not pose any particularly interesting problem.

Let us consider the following class declaration:

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

In FJ, it is not possible to write an expression 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.

In mainstream object-oriented languages, cyclic objects can be indirectly modeled by relying on imperative features. That is, 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.[1] However, method `infOcc` above still does not terminate rather than returning a cyclic object, and in general methods handling cyclic objects must explicitly check cycles to correctly work.

While mainstream object-oriented languages lack any direct support for manipulating cyclic structures, languages like Haskell [12] support potentially infinite data structures by exploiting the expressive power of lazy evaluation [13]. Note that cyclic objects are a very particular case of infinite objects, corresponding to *regular terms* (or trees)[2]: for instance, the list made of infinite occurrences of a given number n is regular, whereas the list of all prime numbers (or, simply, all natural numbers) is not. By lazy evaluation we can express all infinite lists, both regular and non regular. However, the aim is rather different: lazy functions can correctly handle, without looping, all (finite) prefixes of an infinite list (for instance, we can obtain the sum of the first n natural numbers), but cannot handle *whole* cyclic lists. For instance, a lazy function for the previous example `allPos` would not give the correct result, but only all its approximations.

What we propose here is a minimal extension to FJ to support cyclic objects and methods for their direct manipulation. Such a novel programming paradigm is inspired by the recent results concerning the operational semantics of coinductive Prolog [17, 19, 18] 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*=**new** $C(X)$ is an instance of class C whose unique field contains the object itself.
- methods are *regularly corecursive*: if a recursive method call $v.m(\overline{v})$ corresponds to a previous call which is still active on the stack, then then we say that a *coinductive hypothesis* is applied, and such a call terminates immediately.
- moreover, differently from what happens in Prolog, regular corecursion is *controlled*, that is, method declarations have shape $C\ m(\overline{C\ x})$ $\{e$ **with** e'**;**$\}$, where e' is returned in place of e, when a coinductive hypothesis is applied (see the examples below). Standard regular corecursion is obtained when e' is the special variable *res*, hence, in the sequel the syntax $\{e$**;**$\}$ for method bodies is just a shortcut for $\{e$ **with** *res***;**$\}$.

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 3 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.

Let us consider again the example of lists:

```
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))}
}
```

In COFJ one can construct finite lists, as in `new NEList(2,new EList())`, but also cyclic ones, as in `new CycListFact().infOcc(0)`; such a call terminates in COFJ and returns the intended value; since the recursive call is the same as the initial one, a cyclic object is returned, that is, `L = new NEList(0,L)`.

Similarly, we can add to `CycListFact`, method `infAltOcc()` defined as follows:

[1]It is still possible, e.g., in Java, to create instances of `CycList` even when both fields are **final**, but a considerable amount of boilerplate code could be required.

[2]Finitely branching trees whose depth can be infinite, but that can contain only a finite set of subtrees.

```
    NEList infAltOcc(int n1,int n2) {
        new NEList(n1,this.infAltOcc(n2,n1))}
```

Then, `new CycListFact().infAltOcc(1,-1)` returns the following cyclic list

```
L = new NEList(1,new NEList(-1,L)).
```

A method body has shape $\{e$ **with** e'**;**$\}$, where e' denotes the value returned when a coinductive hypothesis is applied, whereas the value denoted by *e* is returned in all other cases. Inside e' the special variable *res* can be used to denote the standard value that would be returned by corecursion.

For instance, as anticipated in the Introduction, the following method correctly works for both non cyclic and cyclic lists:

```
    bool allPos() {
        if(this.el <= 0)
            false
        else
            this.nx.allPos()}
        with
            true
}
```

The same pattern used for `allPos` can be adopted for defining method `member`, but in this case **`false`** is returned when a coinductive hypothesis is applied, as happens (not by coincidence, in this case) in 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}
}
```

To show an example where the default value is not a boolean, we define the method `noRep` which, invoked on a possibly cyclic (that is, infinite) list, returns the corresponding non cyclic (finite) 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() in
            if(l.member(this.el))
                l
            else
                new NEList(this.el,l)
            with
                new EList()
    }
}
```

For brevity we have used the **`let in`** construct, with the standard semantics. Note that, in case `noRep` is invoked on the cyclic list `L = new NEList(0,L)`, the invocation `this.nx.noRep()` in the body of `noRep` would be on exactly the same list, hence, if the **`with`** expression is omitted (hence, **`res`** is returned when a coinductive hypothesis is applied), then the result of `this.nx.noRep()` is the undetermined value, hence the evaluation of the expression `l.member(this.el)` that follows the corecursive invocation would fail (that is, the semantics would be undefined, see the formalization below).

We end this section with two more examples: the former is method `isCyc`, which checks whether a list is cyclic, showing an example where the value returned in the inductive base case is different from the value returned when a coinductive hypothesis is applied.

```
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 positive integers occurring in a list. In particular, if a list is cyclic and contains at least one non positive element, then the method is expected to return a cyclic list. For brevity we only consider the definition of the method in class `NEList`, which is the most interesting case:

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

This naive solution fails to behave correctly in some cases.

For instance, `(L = new NEList(1,new NEList(-1,L))).wrongRemPo` returns the non cyclic list `new NEList(-1,new EList())`, instead of the cyclic list `L = new NEList(-1,L)` (representing the list of infinite occurrences of -1). Indeed, if a list is cyclic, then *res* should be returned, except when the list contains only positive elements; in this last case the empty list has to be returned. Hence, we can correct the code above by using the previously defined method `allPos`.

```
class EList extends List {
    List remPos() { new EList() }
}
class NEList extends List {
    int el; List nx;
    List remPos() {
        if(this.el > 0)
            this.nx.remPos()
        else
            new NEList(this.el,this.nx.remPos())
        with
            if(this.allPos())
                new EList()
            else
                res
    }
}
```

The check `this.allPos()` must be necessarily performed after the coinductive hypothesis has been applied: any earlier attempt is bound to fail, since the cycle may begin at any arbitrary position in the list, and the beginning of the cycle is detected only when the corecursive hypothesis is applied. For instance, if l=`new NEList(-1,L=new` then l`.allPos()` evaluates to **`false`**, while l`.nx.allPos()` evaluates to **`true`**; therefore, as expected, l`.remPos()` returns `new NEList(-1,n`

3. FORMAL DEFINITION

The syntax of COFJ is given in Figure 1. We follow the FJ notations and conventions: we assume infinite sets of *class names* C, including the special class name *Object*, *field names* f, *method names* m, and *variables* x, including the special variables *this* and *res*, and we write $\overline{cd}$ as a shorthand for a possibly empty sequence $cd_1 \dots cd_n$, and analogously for other sequences. The length of a sequence $\overline{x}$ is written $\#\overline{x}$, and the domain and image of a map are written dom and img, respectively.

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

Figure 1: COFJ syntax

Every class has an implicit constructor as in FJ, and the FJ well-formedness conditions on a program p are assumed: names of declared classes are distinct and different from *Object*, hence p can be seen as a map from class names into class declarations s.t. $Object \notin dom(p)$. The inheritance relation (transitive closure of the **`extends`** relation) is acyclic. Method names and field names in a class, and parameter names in a method, are distinct and different from *this* and *res*, and field names declared in a class are distinct from those declared in its superclasses (no field hiding). Finally, for every class name C (except *Object*) occurring in p, we have $C \in dom(p)$.

The syntax deviates from FJ in the following aspects: an infinite set of *labels* X is used, cast expressions have been omitted, method bodies consist of a **with** expression where the special variable *res* can be used in the rhs, and the definition of values is more general.

In our previous proposal [7] corecursion was handled at call rather than at declaration site, thus making the operational semantics more complex, without any apparent gain in expressive power. More importantly, this simplification in the design of the language allowed us to define the type system presented in Section 4.

In FJ values have shape **new** $C(\overline{v})$, that is, are (a concrete representation of) inductive terms built by constructor invocations. Here, values are allowed to be cyclic, 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**=new** $C(\overline{v})$, abbreviated $\overline{X\,\textbf{=}}$ **new** $C(\overline{v})$ with our convention.[3]

We expect the result of evaluating a top-level expression to be *closed*, that is, with all references bound to existing labels. Values corresponding to cyclic objects as X**=new** $C(X)$ are *not* valid expressions, but can be obtained as results of a method invocation, as shown in the examples of previous section. This choice allows us to keep the language minimal; we leave for further work the investigation of more compact linguistic mechanisms for denoting cyclic objects.

Closed values are a concrete representation of regular terms built by constructor invocations, except for the undetermined value which is denoted by all equations having shape X_1**=**$\ldots$ X_n**=**X_i, with $i \in 1..n$. In the semantic rules, all 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 equal, and an analogous assumption holds for open values as well. As a consequence, if the results of two closed expressions e and e' are the same modulo this equivalence, then e and e' can replace each other in any context.

[3] Values annotated with more than one label, like, e.g., `X=Y=`**new** `C(X)`, can be obtained by reduction, see Figure 4.

Open values and the undetermined value 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.

We formalize COFJ in a big-step style for simplicity. In order to distinguish stuck execution from non termination, we use the standard technique of introducing a special wrong result `w`.

The big-step semantics $e, \sigma, \pi \Downarrow r$ returns the result r, if any, of evaluating an expression e in the context of a *call trace* σ, and of a *frame* π defining the values of all local variables (that is, all formal parameters, and the special variables *this* and *res*). The relation should be indexed over programs, however for brevity we leave implicit such an index in all judgments defined in the paper. A call trace is an injective map from expressions of the form $v.m(\overline{v})$, called *(invocation) redexes*, to labels X which represent the corresponding value returned by the method invocation; a frame is a map from variables to values.

Rules without and with error handling are given in Figure 2 and Figure 3, respectively. Some standard technical details have been omitted: the formal definitions of parallel substitution $e[\overline{v}/\overline{x}]$ and the auxiliary functions *fields* and $mbody$. We write $\overline{e}, \sigma \Downarrow \overline{v}$ as a shorthand for the set of judgments $e_1, \sigma \Downarrow v_1 \ldots e_n, \sigma \Downarrow v_n$.

Rule (FIELD) models field access. Recall that, with the FJ convention, $\overline{C\ f\textbf{;}}$ stands for $C_1\ f_1\textbf{;}\ \ldots\ C_n\ f_n\textbf{;}$. The receiver expression is evaluated, and its result is expected to be an object. The standard FJ function *fields* retrieves the sequence of the fields of its class, starting from those inherited, 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 **`class`** `C` **`extends`** `Object { C f; }` if $v =$ `X=new C(Y=new C(X))`, then v`.f` is reduced to

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

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

There are two rules for method invocation. In both, the receiver and argument expressions are evaluated first to obtain the invocation redex $v.m(\overline{v})$. Then, the behaviour is different depending whether a cycle is detected in the call trace σ.

If this is not the case, then the method invocation is handled as usual (rule (INVK)): 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 lhs expression e' of **with** where the receiver object replaces *this* and the arguments replace the parameters. Evaluation of e' is performed in the call trace σ updated with the redex corresponding to the current invocation, associated with a fresh label X. Finally, when the evaluation of the method body is completed, references to the label X in the resulting value (due to termination by coinduction of the method, see (COREC)) are bound. In this way a cyclic object can be obtained as the result of a method invocation.

Rule (COREC) is applied when the method terminates corecursively, that is, a cycle in σ is detected; the rhs expression e' of **with** in the body of the method is evaluated in the new frame where *this* is associated with the receiver object, *res* is associated with the label X found in the call trace, and the formal parameters are associated with the arguments.

For instance, given the classes

```
class C extends Object { Object f; }
class A extends Object {
```

$$\text{(PROG)}\ \frac{e, \emptyset, \emptyset \Downarrow v}{\overline{cd}\ e \Downarrow v} \qquad \text{(VAR)}\ \frac{}{x, \sigma, \pi \Downarrow v}\ \pi(x) = v \qquad \text{(FIELD)}\ \frac{e, \sigma, \pi \Downarrow v}{e.f, \sigma, \pi \Downarrow v_i[v/\overline{X}]}\ \begin{array}{l} v = \overline{X =} \textbf{new}\ C(\overline{v}) \\ \mathit{fields}(C) = \overline{C\ f\texttt{;}} \\ f = f_i, i \in 1..n \end{array}$$

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

$$\text{(COREC)}\ \frac{e, \sigma, \pi \Downarrow v \quad \overline{e}, \sigma, \pi \Downarrow \overline{v} \quad e', \sigma, [\mathit{this}{:}v, \mathit{res}{:}X, \overline{x{:}v}] \Downarrow r}{e.m(\overline{e}), \sigma, \pi \Downarrow r}\ \begin{array}{l} v = \overline{X =} \textbf{new}\ C(_) \\ mbody(C, m) = (\overline{x}, _\ \textbf{with}\ e') \\ \sigma(v.m(\overline{v})) = X \end{array}$$

$$\text{(NEW)}\ \frac{\overline{e}, \sigma, \pi \Downarrow \overline{v}}{\textbf{new}\ C(\overline{e}), \sigma, \pi \Downarrow \textbf{new}\ C(\overline{v})}\ \#\mathit{fields}(C) = \#\overline{e}$$

Figure 2: COFJ big-step rules (without error handling)

$$\text{(W-FIELD)}\ \frac{e, \sigma, \pi \Downarrow v}{e.f, \sigma, \pi \Downarrow \texttt{w}}\ \begin{array}{l} \neg guarded(v) \text{ or} \\ v = \overline{X =} \textbf{new}\ C(_) \text{ and} \\ \mathit{fields}(C) = \overline{C\ f\texttt{;}} \text{ and } f \neq f_i \text{ for all } i \in 1..n \end{array}$$

$$\text{(W-INVK)}\ \frac{e, \sigma, \pi \Downarrow v}{e.m(\overline{e}), \sigma, \pi \Downarrow \texttt{w}}\ \begin{array}{l} \neg guarded(v) \text{ or} \\ v = \overline{X =} \textbf{new}\ C(_) \text{ and} \\ (mbody(C, m) \text{ undefined or} \\ \quad mbody(C, m) = (\overline{x}, e) \text{ and } \#\overline{x} \neq \#\overline{e}) \end{array}$$

$$\text{(W-INVK2)}\ \frac{e, \sigma, \pi \Downarrow v \quad \overline{e}, \sigma, \pi \Downarrow \overline{v} \quad e', \sigma[v.m(\overline{v}){:}X], [\mathit{this}{:}v, \overline{x{:}v}] \Downarrow \texttt{w}}{e.m(\overline{e}), \sigma, \pi \Downarrow \texttt{w}}\ \begin{array}{l} v = \overline{X =} \textbf{new}\ C(_) \\ mbody(C, m) = (\overline{x}, e'\ \textbf{with}\ _) \\ v.m(\overline{v}) \notin dom(\sigma) \\ X \text{ fresh} \end{array}$$

$$\text{(W-NEW)}\ \frac{}{\textbf{new}\ C(\overline{e}), \sigma, \pi \Downarrow \texttt{w}}\ \#\mathit{fields}(C) \neq \#\overline{e} \qquad \text{(PROP)}\ \frac{e, \sigma, \pi \Downarrow \texttt{w}}{\mathcal{C}[e], \sigma, \pi \Downarrow \texttt{w}}\ \mathcal{C}[\,] \neq [\,]$$

Figure 3: COFJ big-step rules for error handling

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

`new A().m1()` is reduced to the cyclic object `X=Y=new C(X)` (equivalent to `X=new C(X)`) as shown in Figure 4.

If method `m1` were `C m1() {this.m2()) with new A();}` then the proof

$$\text{(VAR)}\frac{}{res, \sigma_2, \pi[res{:}X] \Downarrow X}$$

would be replaced by the proof

$$\text{(NEW)}\frac{}{\textbf{new A}(), \sigma_2, \pi[res{:}X] \Downarrow \textbf{new A}()}$$

and `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. The side condition ensures that the constructor is invoked with the appropriate number of arguments.

Rules (W-FIELD), (W-INVK) and (W-NEW) model the cases in which field access, method invocation, and constructor invocation, respectively, cannot be performed. The predicate $guarded(v)$ holds whenever v is an object, that is, of shape $\overline{X} = \textbf{new } C(\overline{v})$.

Field access fails if the receiver is not an object (first alternative in the side condition) or it is an instance of a class which does not provide a field with the required name (second alternative). Analogously, method invocation fails if the receiver is not an object, or it is an instance of a class which either does not provide a method with the required name, or it provides a method with a wrong number of parameters. Constructor invocation fails if the constructor has a wrong number of parameters.

Rules (PROP) and (W-INVK2) model propagation of the `w` result. The former handles standard contextual propagation, whereas the latter handles the case when a method invocation fails since the execution of the corresponding method body fails; while rule (COREC) in Figure 2 includes also error propagation (by simply using the meta-variable r), for rule (INVK) the extra rule (W-INVK2) is needed, since X=`w` is not a syntactically valid value.

We show the consistency of the calculus by the following two theorems. The former states that the evaluation of an expression returns, if any, a value whose free labels are defined in the call trace (hence, in particular, if the call trace is empty, then the returned value is closed). The latter states that COFJ semantics conservatively extends the FJ semantics, that is, if we get a result by FJ semantics, then we get the same result by the COFJ 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.

THEOREM 3.1. *If* $e, \sigma, \pi \Downarrow v$, *then* $FL(v) \subseteq img(\sigma)$.

PROOF. By induction on the rules defining $e, \sigma, \pi \Downarrow v$, which are those in Figure 2.

- (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.
- (NEW): trivially by inductive hypothesis. □

$$
\begin{array}{lll}
T & ::= & C^g \\
g & ::= & + \mid - \\
fd & ::= & T\, f\,\texttt{;} \\
md & ::= & U_1 \textbf{ with } U_2\, m(\overline{T\ x})\ \{e \textbf{ with } e'\texttt{;}\,\} \\
U & ::= & T \mid T? \\
\Gamma & ::= & \overline{x{:}U}
\end{array}
$$

Figure 5: COFJ types and type environments

The standard syntax and recursive semantics $e \Downarrow_{\text{FJ}} r$ of FJ in big-step style are reported in the Appendix.

THEOREM 3.2. *For e expression and r result in* FJ, *if* $e \Downarrow_{\text{FJ}} r$, *then* $e, \sigma, \emptyset \Downarrow r$ *for all* σ.

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

- (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.
- Every remaining rule is exactly analogous, or can be obtained as special instance, of the corresponding COFJ rule, hence the thesis trivially holds.

□

4. TYPE SYSTEM

We present a compositional type system which is an extension of the standard nominal type system of FJ, able to statically distinguish expressions whose values are guaranteed to be closed and different from the undetermined value, from those that may evaluate to the undetermined value or to an open value.

Types and type environments for COFJ are defined in Figure 5.

In COFJ a *closed* type T is a standard FJ nominal type C tagged by either '$+$', or '$-$': C^+ specifies all closed values of type C (and its subclasses) other than the undetermined value, whereas C^- is a proper supertype of C^+ that includes also the undetermined value (but not the open values).

An *open* type $T?$ is associated with those expressions whose values are possibly open, but will eventually evolve to closed values of type T; such values originate from the evaluation of the special variable *res* in the rhs of **with**. The type associated with *res* is always of shape $C^-?$; this is a conservative assumption, since if the open value associated with *res* is not guarded by a constructor in the lhs e of **with**, then the closed value that will be eventually returned after evaluating e will be undetermined, hence the correct type of the returned value is C^-; however, if the open value associated with *res* is always guarded by a constructor, then the type of the closed value that will be finally returned is allowed to be C^+.

The meta-variable U denotes either a closed or an open type.

The syntax of the typed language is specified in Figure 5): field and method declarations are annotated with types. In particular, for ensuring the soundness of the system, fields and formal parameters can only be annotated with closed types. The type of a value returned by a method is specified by the pair U_1 **with** U_2, where U_1 and U_2 are derived from the lhs and rhs part, respectively, of the **with** expression. The subtyping relation $U_1 \leq U_2$ must be always satisfied to guarantee soundness; the two return types are used in

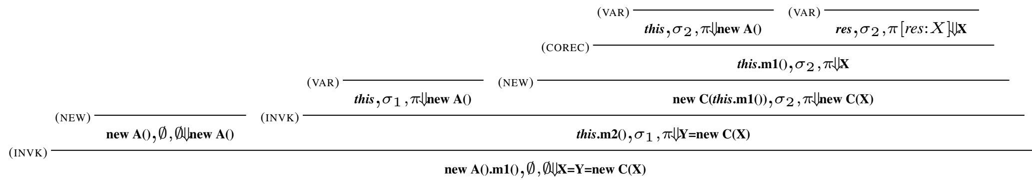

Figure 4: Example of reduction, where $\sigma_1 = [\text{new A().m1()}{:}\text{X}]$, $\sigma_2 = \sigma_1[\text{new A().m2()}{:}\text{Y}]$, $\pi = [\textit{this}{:}\text{new A()}]$

$$\text{(SUB-TAGGED)}\ \frac{C_1 \leq C_2 \quad g_1 \leq g_2}{C_1^{g_1} \leq C_2^{g_2}} \qquad \text{(SUB-TAG)}\ \frac{g_1 = + \vee g_2 = -}{g_1 \leq g_2}$$

$$\text{(EMB-FROZEN)}\ \frac{}{T \leq T?} \qquad \text{(SUB-FROZEN)}\ \frac{T_1 \leq T_2}{T_1? \leq T_2?}$$

Figure 6: COFJ subtyping rules

different contexts: U_1 is only used for top-level method invocations contained in the main expression *e* of the program; in this case the returned value is always closed, because all invocations necessarily originate from *e*. In all other cases the less specific type U_2 is used, corresponding to the conservative assumption that the returned value may be open. This approach can be overly conservative in some cases; however, a more accurate static analysis could be employed to partition methods into strata of mutually recursive methods. This would allow a more permissive typing rule for method invocation in case the invoked method belongs to a different stratum.

Finally, a type environment Γ is a finite map from variables (including the special variables *this* and *res*) to types U.

The straightforward subtyping rules are defined in Figure 6; we have omitted the standard definition of nominal subtyping between class names. The following chain of subtyping relation holds for any class C: $C^+ \leq C^+? \leq C^-? \leq C^-$. All pairs are intuitive except for the last $C^-? \leq C^-$ that can be explained by the fact that the closed value which an open value will eventually evolve to, may be undetermined or not, depending on the context. For instance, the open value *X* can evolve either to $X = X$ (undetermined, type C^-) or to $X = \textbf{new}\ C(X)$ (determined, type C^+). The undetermined value is a closed value that cannot evolve, and, therefore, it will always be undetermined (see rules (T-NEW1) and (T-NEW2)).

Typing rules significantly deviates from FJ and are defined in Figure 7. For brevity we leave implicit the dependency of all judgments from the enclosing program.

The typing judgment for expressions has shape $g; \Gamma \vdash e : T$, where g is a tag that indicates the context where a method is invoked: $+$ corresponds to the main expression of the program, whereas $-$ specifies any other context (that is, any method body); Γ is the type environment, *e* is the expression to be typed, and T is its corresponding type.

Typechecking a program corresponds to typecheck all its class declarations, and its main expression. This is the only case where an expressions is typechecked with tag $+$ because at top-level it is always safe assuming that the value returned by a method invocation is closed.

Typechecking for class declarations is standard, and is defined on top of the typing judgment for method declarations which depends on the class where the method is declared.

A method declaration is well-typed if the type annotations of the method are respected. Both the lsh and rhs of the **with** expression are typechecked with tag $-$, since there is no guarantee that the returned value is closed. The type environment for both *e* and *e′* assigns the type C^+ to *this*, where C is the class containing the method to be checked; indeed, method invocations are type safe only if the expression denoting the target object has type C^+ (see rule T-INVK); furthermore, the Γ contains all formal parameters with their corresponding declared types. Finally, for expression *e′* only, Γ contains also the special variable *res*; its type is conservatively assumed to be $C'^-?$, where C' is the underlying class name of the type U_1 of the expression *e*. The lhs return type U_1' must be a supertype of the type $U_1\Downarrow$ obtained by removing (if present) the ? constructor from U_1 (see the definition of the $_\Downarrow$ operator at the bottom); recall that the lhs return type can be used only under the assumption that the value returned by the method is closed.

The side condition $ok_override(C, m)$ is the standard check on method overriding as defined in FJ (for this reason the definition of $ok_override(C, m)$ has been omitted), while the condition $U_1' \leq U_2'$ ensures that the rhs return type is always a conservative approximation of the corresponding lhs type.

Rule (T-VAR) for variables is standard; rule (T-FIELD) states that field selection is type safe only if *e* has type C^+, that is, *e* cannot evaluates to the undetermined value, neither to an open value. Except for this, the rule is the same as the corresponding FJ rule.

As happens for rule (T-FIELD), rule (T-INVK) requires the expression e_0 denoting the target object of a method invocation to have type C^+, to avoid that *e* could evaluate to the undetermined value or to an open value. The returned type depends on the context, specified by the tag g, where the expression is typechecked: if $g = +$, then the lhs return type is considered, otherwise the lhs is taken. All other checks are standard, as well as the auxiliary function $mtype$ (whose definition has been omitted) returning the type of method *m* of class C.

For object creation two different typing rules are provided. Rule (T-NEW1) is applicable when no argument has an open type; in this case the resulting type of the expression is C^+. However, if some argument has an open type, then rule (T-NEW2) can be applied in place of (T-NEW1), in this case the types of all arguments can be narrowed by means of the operator $_\Downarrow_+$ (defined after the typing rules); narrowing has effect only on open types, and convert them in closed types tagged with $+$. This is sound because in COFJ constructors it is not possible to access the field or to invoke the method of an object passed as an argument. The returned type of the whole expression is the open type $C^+?$ because there is no guarantee that the corresponding value is closed, since there could be some pending method invocation[4] that still needs to be completed; however, at the top-level all method invocations will be completed and the value will be closed, hence its type will be C^+ (recall the side con-

[4] $X_1 = \textbf{new}\ C(X_2)$ (with $X_1 \neq X_2$) is an example of value of type $C^+?$.

$$\text{(T-PROG)}\ \frac{\vdash cd_1 \ldots \vdash cd_n \quad +;\emptyset \vdash e : U}{\vdash \overline{cd}\ e} \qquad \text{(T-CDEC)}\ \frac{C \vdash md_1 \ldots C \vdash md_n}{\vdash \textbf{class}\ C\ \textbf{extends}\ C'\ \{\ \overline{fd}\ \overline{md}\ \}}$$

$$\text{(T-MDEC)}\ \frac{\begin{array}{l} -;\mathit{this}{:}C^+, \overline{x{:}T} \vdash e : U_1 \\ -;\mathit{this}{:}C^+, \mathit{res}{:}\mathit{class}(U_1)^-?, \overline{x{:}T} \vdash e' : U_2 \end{array}}{C \vdash U_1'\ \textbf{with}\ U_2'\ m(\overline{T\ x})\ \{e\ \textbf{with}\ e' \textbf{;}\}} \quad \begin{array}{l} U_1{\Downarrow} \leq U_1', U_2 \leq U_2', U_1' \leq U_2' \\ \mathit{ok_override}(C, m) \end{array}$$

$$\text{(T-VAR)}\ \frac{}{g;\Gamma \vdash x : U}\ \Gamma(x) = U \qquad \text{(T-FIELD)}\ \frac{g;\Gamma \vdash e : C^+}{g;\Gamma \vdash e.f : T_i}\ \begin{array}{l} \mathit{fields}(C) = \overline{T\ f;} \\ f = f_i, i \in 1..n \end{array}$$

$$\text{(T-INVK)}\ \frac{g;\Gamma \vdash e_0 : C^+ \quad g;\Gamma \vdash \overline{e} : \overline{T}'}{g;\Gamma \vdash e_0.m(\overline{e}) : U'} \quad \begin{array}{l} \mathit{mtype}(C, m) = \overline{T} \rightarrow T\ \textbf{with}\ U \\ \overline{T'} \leq \overline{T} \\ U' = \begin{cases} T & \text{if } g = + \\ U & \text{if } g = - \end{cases} \end{array}$$

$$\text{(T-NEW1)}\ \frac{g;\Gamma \vdash \overline{e} : \overline{T'}}{g;\Gamma \vdash \textbf{new}\ C(\overline{e}) : C^+}\ \begin{array}{l} \mathit{fields}(C) = \overline{T\ f;} \\ \overline{T'} \leq \overline{T} \end{array} \qquad \text{(T-NEW2)}\ \frac{g;\Gamma \vdash \overline{e} : \overline{U}}{g;\Gamma \vdash \textbf{new}\ C(\overline{e}) : C^+?}\ \begin{array}{l} \mathit{fields}(C) = \overline{T\ f;} \\ \overline{U}{\Downarrow_+} \leq \overline{T} \end{array}$$

$$T{\Downarrow} = T \qquad T?{\Downarrow} = T \qquad T{\Downarrow_+} = T \qquad C^g?{\Downarrow_+} = C^+ \qquad \mathit{class}(C^g) = C$$

Figure 7: COFJ typing rules

dition $U_1{\Downarrow} \leq U_1'$ in rule (T-MDEC)). In this way, the type system prevents field selection, method invocation and argument passing (to methods, but not to constructors) of open values.

To better explain how the type system works, we show few examples of typings. We start with the following simple class declaration (assuming that class `C` is defined in the same program):

```
class H extends Object {
  C⁻ with C⁻ m() { this.m() with res; }
}
```

According to rule (T-MDEC) we have $-;\texttt{this}{:}\texttt{H}^+ \vdash \texttt{this.m}() : \texttt{C}^-$ and $-;\texttt{this}{:}\texttt{H}^+, \texttt{res}{:}\texttt{C}^-? \vdash \texttt{res} : \texttt{C}^-?$; because $\texttt{C}^-? \leq \texttt{C}^-$, the only return type derivable for the method is `C⁻ with C⁻` (by the side condition $U_1' \leq U_2'$ of rule (T-MDEC)) for any class `C` defined in the program.

Let us consider the following variation of class `H` in a program where class `C` has just one field, and its type is C^+:

```
class H extends Object {
  C⁺ with C⁻? m() { new C(this.m()) with res; }
}
```

The class is well-typed thanks to rule (T-NEW2); indeed, we have $-;\texttt{this}{:}\texttt{H}^+ \vdash \texttt{new C}(\texttt{this.m}()) : \texttt{C}^+?$, therefore we can derive the return type `C⁺ with C⁻` (by the side condition $U_1{\Downarrow} \leq U_1'$ of rule (T-MDEC)).

Let us now consider a class that cannot be typed.

```
class C extends Object {
  U f;
  U with C⁻? m() { new C(this.m().f) with res; }
}
```

Independently from the type U, we have $-;\texttt{this}{:}\texttt{H}^+ \vdash \texttt{this.m}() : \texttt{C}^-?$, hence `this.m().f` cannot be correctly typed according to rule (T-FIELD).

Assuming to replace primitive types `int` and `bool` with classes implementing the standard encoding of these types with objects, the example in Section 1 can be typed with the following type annotations:

```
class RepDecFact extends Object {
   RepDec⁺ with RepDec⁻? zero() {
      new RepDec(0,zero()) with res
   }
}
class RepDec extends Object {
    Int⁺ digit;
    RepDec⁺ next;
    RepDec⁺ with RepDec⁻? compl() {
        new RepDec(9-this.digit,this.next.compl())
        with res
    }
    Bool⁺ with Bool⁺ isZero() {
       digit==0 && this.next.isZero() with true
    }
}
```

We conclude this section by stating the soundness claim.

THEOREM 4.1. *If* $\emptyset \vdash e : T$, *and* $e, \emptyset, \emptyset \Downarrow r$, *then* $r \neq \mathtt{w}$.

5. RELATED WORK AND CONCLUSION

This paper represents a further step towards the integration of the object-oriented paradigm with coinductive programming, a promising sub-paradigm originating from logic programming, and expressly devised to ease high-level programming and reasoning with cyclic data structures. More precisely, we have enhanced our previous proposal by defining a simpler construct for dealing with regular corecursion, and, consequently, a cleaner operational semantics. More importantly, such a simplification has allowed us to define a type system able to conservatively prevent *unsafe* use of spurious values (that is, open values and the undetermined value) that may be returned by corecursive methods.

This paper is inspired by recent work on coinductive logic programming and regular recursion in Prolog. Simon et al. [17, 19, 18] 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 [19]. Coinductive logic programming has proved to be useful for formal verification [14, 15], 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 [21, 11], 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 [20, 10, 16].

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, this paper is more focused on the challenge of extending object-oriented languages to support coinductive programming.

There exist several interesting directions for further research on the integration of coinductive programming with the object-oriented paradigm. On the foundational side, it would be important to explore techniques to prove the correctness of corecursive methods, possibly integrated with proof assistants, as Coq [9], that provide built-in support for coinductive definitions and proofs by coinduction.

On the more practical side, although the proposed type annotations are not particularly heavy, an inference algorithm able to derive part of them would be useful; furthermore, as already mentioned in Section 4, a more accurate analysis on mutual dependencies between methods would allow a more permissive type system. Another important issue is the extension of the semantics of corecursive methods to the imperative setting, and the study of a corresponding effective implementation.

6. 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. An extended version submitted for journal publication is available at `ftp://ftp.disi.unige.it/person/AnconaD/AnconaExtendedSAC12.pdf`.

[3] D. Ancona. Soundness of object-oriented languages with coinductive big-step semantics. In *ECOOP 2012*, pages 459–483, 2012.

[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] D. Ancona and E. Zucca. Corecursive Featherweight Java. In *FTfJP '12*, 2012.

[8] Davide Ancona and Elena Zucca. Translating corecursive Featherweight Java in coinductive logic programming. In *Co-LP 2012 - A workshop on Coinductive Logic Programming*, 2012.

[9] 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.

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

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

[12] 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.

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

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

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

[16] 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.

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

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

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

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

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

APPENDIX

A. FJ FORMAL DEFINITION

2. PROGRAMMING WITH COFJ

We present some more significant examples of COFJ programming, that show the usefulness of regular terms and controlled regular corecursion.

2.1 Finite automata and regular languages

We 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;
}
```

$$
\begin{array}{lcl}
e & ::= & x \mid e.f \mid e.m(\overline{e}) \mid \textbf{new}\ C(\overline{e}) \\
u, v & ::= & \textbf{new}\ C(\overline{v}) \\
r & ::= & v \mid \mathtt{W} \\
\mathcal{C}[\,] & ::= & [\,] \mid \mathcal{C}[\,].f \mid \mathcal{C}[\,].m(\overline{e}) \mid e.m(\overline{e}, \mathcal{C}[\,], \overline{e}') \mid \textbf{new}\ C(\overline{e}, \mathcal{C}[\,], \overline{e}')
\end{array}
$$

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

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

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

$$
\text{(FJ-W-FIELD)}\ \frac{e \Downarrow_{\mathrm{FJ}} v}{e.f \Downarrow_{\mathrm{FJ}} \mathtt{W}} \quad \begin{array}{l} v = \textbf{new}\ C(_) \\ \mathit{fields}(C) = \overline{C\ f;} \\ f \neq f_i \text{ for all } i \in 1..n \end{array}
$$

$$
\text{(FJ-W-INVK)}\ \frac{e \Downarrow_{\mathrm{FJ}} v}{e.m(\overline{e})\ [\textbf{with}\ _] \Downarrow_{\mathrm{FJ}} \mathtt{W}} \quad \begin{array}{l} v = \textbf{new}\ C(_) \\ \mathit{mbody}(C, m) \text{ undefined or} \\ \quad \mathit{mbody}(C, m) = (\overline{x}, _) \text{ and} \\ \quad \#\overline{x} \neq \#\overline{e} \end{array}
$$

$$
\text{(FJ-PROP)}\ \frac{e \Downarrow_{\mathrm{FJ}} \mathtt{W}}{\mathcal{C}[e] \Downarrow_{\mathrm{FJ}} \mathtt{W}} \quad \mathcal{C}[\,] \neq [\,]
$$

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

Figure 8: FJ syntax and big-step rules

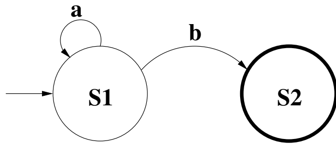

Figure 9: A deterministic finite automaton recognizing the language `a*b`

Let us consider the automaton depicted in Figure 9, 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.[5]

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

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

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
    }
}
```

2.2 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 digit; RepDec next; }
```

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))
```

In Section 1 we have already considered examples of regularly coinductive methods with repeating decimals.

```
class RepDecFact extends Object {
   RepDec zero() { // returns the repeating decimal 0
      new RepDec(0,zero()) with res
   }
}
class RepDec extends Object {
    int digit;
    RepDec next;
    RepDec compl() { // returns 1 - this
        new RepDec(9-this.digit,this.next.compl())
        with res
    }
    bool isZero() { // check if this is zero
       (digit==0 && this.next.isZero()) with true
    }
}
```

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 auxiliary method that computes the carry of the addition of two repeating decimals (where `/` denotes integer division).

```
class RepDec extends Object {
   int digit; RepDec next;
   ...
   int carry(RepDec d) {
      (this.digit + d.digit + this.next.carry(d.next)) / 10
      with 0
   }
}
```

By regularity of **`this`** and `d`, method `carry` is always guaranteed to terminate corecursively; when a cycle in the call trace is detected value 0 is returned; this is correct even when the globally computed carry must be 1, because such a carry necessarily generates from a pair of digits whose addition is strictly greater than 9 and must be propagated to the most significant position. For instance, if `d1` and `d2` represent $0.\overline{8}$ and $0.\overline{121}$, respectively, then `d1.carry(d2)` yields 1, as expected.

Method `add` is simply defined in terms of `carry` (where `%` denotes the reminder operator):

```
class RepDec extends Object {
   int digit; RepDec next;
   ...
   RepDec add(RepDec d) {
      new RepDec( (this.digit +
                   d.digit +
                   this.next.carry(d.next)
                  ) % 10,
                 this.next.add(d.next))
      with res
   }
}
```

As expected, the result of the addition is a new repeated decimal where the most significant digit is obtained by computing the reminder by 10 of the addition of the most significant digits of the two operands increased by the carry returned by the addition of the remaining part of the two operands, and where the rest of the digits are obtained by adding the remaining parts of the two operands.

Again, by regularity of **`this`** and `d`, method `add` is always guaranteed to terminate corecursively; differently from method `carry`, the returned value is guarded, therefore the rhs of **`with`** is **`res`**.

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 $0.4\overline{9}$.

2.3 Graphs

Graphs are perhaps the most interesting application domain of controlled 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 controlled 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.