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.

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 —*Type structure*

[*]This work has been partially supported by MIUR CINA (Compositionality, Interaction, Negotiation, Autonomicity for the future ICT society)

General Terms

Languages, Theory

Keywords

coinduction, programming paradigms, regular terms, Java

1. INTRODUCTION

In previous work we have presented COFJ [6], an extension to Featherweight Java (FJ) 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 considerthe following COFJ classes.

```
   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() { // checks 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 instance, 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 2, **`res`** is a special variable (like **`this`**) which denotes a well-defined value only in case the generated recursive equations admits 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 recursive 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 literal **`true`** has to be returned.

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 give the formal definition of COFJ, in Section 3 the type system and the related soundness result. Finally, in Section 4 we outline related and further work.

In [7], we have provided a derived semantics of COFJ by translation into coinductive logic programming.

2. 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 $cd_1 \ldots cd_n$, 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.

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 $C \in dom(p)$.

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

Figure 1: COFJ syntax

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 [6] 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 3.

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. Values which are not labels, that is, of shape X_1**=**$\ldots$ X_n**=new** $C(\overline{v})$, abbreviated $\overline{X\textbf{=}}$ **new** $C(\overline{v})$ with our convention, are *objects*.[1]

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 previous examples. 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:

```
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.

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.

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`.

[1]Values annotated with more than one label, like, e.g., `X=Y=`**new** `C(X)`, can be obtained by reduction, see Figure 5.

If $C = \textbf{class}\ C\ \textbf{extends}\ C'\ \{\ \overline{fd}\ md_1 \ldots md_n\ \}$ then
$\mathit{fields}(C) = \mathit{fields}(C')\ \overline{fd}$,

$$mbody(C,m) = \begin{cases} (\overline{x}, e\ \textbf{with}\ e') & \text{if } md_i = C\ m(\overline{C\ x})\ \{e\ \textbf{with}\ e'\,\mathtt{;}\} \\ & \text{for some } i \in 1..n \\ mbody(C',m) & \text{otherwise} \end{cases}$$

$\mathit{fields}(\mathit{Object}) = \Lambda$, $mbody(\mathit{Object}, m)$ undefined for all m.

$v[u/X\,\overline{X}] = v[u/X][u/\overline{X}]$, if $\overline{X} \neq \Lambda$
$(\textbf{new}\ C(v_1,\ldots,v_n))[v/X] = \textbf{new}\ C(v_1[v/X],\ldots,v_n[v/X])$

$$(X\mathbf{=}v)[u/Y] = \begin{cases} v & \text{if } X\text{=}Y \\ v[u/X] & \text{otherwise} \end{cases} \qquad X[v/Y] = \begin{cases} v & \text{if } X\text{=}Y \\ X & \text{otherwise} \end{cases}$$

Figure 4: Auxiliary functions

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 *(call) redexes*, to labels X which represent the 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. We write $\overline{e}, \sigma \Downarrow \overline{v}$ as a shorthand for the set of judgments $e_1, \sigma \Downarrow v_1 \ldots e_n, \sigma \Downarrow v_n$. We use the standard auxiliary functions defined in Figure 4.

Rule (FIELD) models field access. Recall that, with the FJ convention, $\overline{C\ f\,\mathtt{;}}$ stands for $C_1\ f_1\mathtt{;}\ \ldots\ C_n\ f_n\mathtt{;}$. 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 $\mathit{fields}(\texttt{C}) =$ `C f;` and $(\texttt{Y=new C(X)})[v/\texttt{X}] = u$.

There are two rules for method invocation. In both, the receiver and argument expressions are evaluated first to obtain the call redex $v.m(\overline{v})$. Then, the behavior 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 A extends Object {
  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 5.

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

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

would be replaced by the proof

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

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 $\overline{X\mathbf{=}}\ \textbf{new}\ C(\overline{v})$.

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.

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

For space limitation, the standard syntax and semantics $e \Downarrow_{\text{FJ}} r$ of FJ in big-step style have been omitted.

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

3. TYPE SYSTEM

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

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

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

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

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

$$\textsc{(w-field)}\ \frac{e,\sigma,\pi \Downarrow v}{e.f,\sigma,\pi \Downarrow \texttt{w}}\ \begin{array}{l} \neg \mathit{guarded}(v)\ \text{or} \\ v=\overline{X=}\,\textbf{new}\ C(\_)\ \text{and} \\ \mathit{fields}(C)=\overline{C\ f\texttt{;}}\ \text{and}\ f\neq f_i\ \text{for all}\ i\in 1..n \end{array} \qquad \textsc{(w-invk)}\ \frac{e,\sigma,\pi \Downarrow v}{e.m(\overline{e}),\sigma,\pi \Downarrow \texttt{w}}\ \begin{array}{l} \neg \mathit{guarded}(v)\ \text{or} \\ v=\overline{X=}\,\textbf{new}\ C(\_)\ \text{and} \\ (\mathit{mbody}(C,m)\ \text{undefined or} \\ \quad \mathit{mbody}(C,m)=(\overline{x},e)\ \text{and}\ \#\overline{x}\neq\#\overline{e}) \end{array}$$

$$\textsc{(w-invk2)}\ \frac{e,\sigma,\pi \Downarrow v \quad \overline{e},\sigma,\pi \Downarrow \overline{v} \quad e',\sigma[v.m(\overline{v}){:}X],[\mathit{this}{:}v,\overline{x}{:}\overline{v}] \Downarrow \texttt{w}}{e.m(\overline{e}),\sigma,\pi \Downarrow \texttt{w}}\ \begin{array}{l} v=\overline{X=}\,\textbf{new}\ C(\_) \\ \mathit{mbody}(C,m)=(\overline{x},e'\ \textbf{with}\ \_) \\ v.m(\overline{v})\notin \mathit{dom}(\sigma) \\ X\ \text{fresh} \end{array}$$

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

Figure 3: COFJ big-step rules for error handling

$$\textsc{(invk)}\ \dfrac{\textsc{(new)}\ \dfrac{}{\textbf{new}\ \texttt{A}(),\emptyset,\emptyset \Downarrow \textbf{new}\ \texttt{A}()} \quad \textsc{(invk)}\ \dfrac{\textsc{(var)}\ \dfrac{}{\mathit{this},\sigma_1,\pi \Downarrow \textbf{new}\ \texttt{A}()} \quad \textsc{(new)}\ \dfrac{\textsc{(corec)}\ \dfrac{\textsc{(var)}\ \dfrac{}{\mathit{this},\sigma_2,\pi \Downarrow \textbf{new}\ \texttt{A}()} \quad \textsc{(var)}\ \dfrac{}{\mathit{res},\sigma_2,\pi[\mathit{res}{:}\texttt{X}] \Downarrow \texttt{X}}}{\mathit{this}.\texttt{m1}(),\sigma_2,\pi \Downarrow \texttt{X}}}{\textbf{new}\ \texttt{C}(\mathit{this}.\texttt{m1}()),\sigma_2,\pi \Downarrow \textbf{new}\ C(\texttt{X})}}{\mathit{this}.\texttt{m2}(),\sigma_1,\pi \Downarrow \texttt{Y=}\textbf{new}\ \texttt{C}(\texttt{X})}}{\textbf{new}\ \texttt{A}().\texttt{m1}(),\emptyset,\emptyset \Downarrow \texttt{X=Y=}\textbf{new}\ \texttt{C}(\texttt{X})}$$

$\sigma_1=[\textbf{new}\ \texttt{A}().\texttt{m1}(){:}X]$, $\sigma_2=\sigma_1[\textbf{new}\ \texttt{A}().\texttt{m2}(){:}\texttt{Y}]$,$\pi=[\mathit{this}{:}\textbf{new}\ \texttt{A}()]$

Figure 5: Example of reduction

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

Figure 6: COFJ types and type environments

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 6.

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 6): 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 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 is too conservative in some cases; however, with a more accurate static analysis strata of mutually recursive methods could be identified. 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 subtyping rules are defined in Figure 7; 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 $X =$ **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)).

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

$$
\text{(OPEN+)}\ \frac{C^+ \leq T}{C^+ \leq T?} \qquad \text{(OPEN}-\text{)}\ \frac{T \leq C^-}{T? \leq C^-}
$$

$$
\text{(SUB-OPEN)}\ \frac{T_1 \leq T_2}{T_1? \leq T_2?}
$$

Figure 7: COFJ subtyping rules

The following properties hold for the subtyping relation. Lemmas 3.1, and 3.2 are instrumental to lemma 3.3.

LEMMA 3.1. *The subtyping relation is transitive.*

LEMMA 3.2. *The type operators $\Downarrow$ and $\Downarrow_+$ preserve subtyping.*

LEMMA 3.3. *If $\overline{U'} \leq \overline{U}$, and $g; \overline{x{:}U} \vdash e : U$, then $g; \overline{x{:}U'} \vdash e : U'$, with $U' \leq U$.*

Typing rules significantly deviates from FJ and are defined in Figure 8. For brevity we leave implicit the dependency of all judgments on the enclosing program.

The typing judgment for expressions has shape $g; \Gamma \vdash e : T$, where g indicates the context where a method is invoked: $+$ corresponds to the main expression of the program, whereas $-$ specifies 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 the value returned by a method invocation is always 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 (the definition has been omitted for brevity), 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* can-

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

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

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

$$\text{(T-INVK)}\ \frac{g;\Gamma \vdash e_0 : C^+ \quad g;\Gamma \vdash \overline{e} : \overline{U'}}{g;\Gamma \vdash e_0.m(\overline{e}) : U'} \quad \begin{array}{l} mtype(C,m) = \overline{T} \rightarrow U_1\ \textbf{with}\ U_2 \\ \overline{U'} \leq \overline{T} \\ U' = \begin{cases} U_1 & \text{if } g = + \\ U_2 & \text{if } g = - \end{cases} \end{array}$$

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

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

Figure 8: COFJ typing rules

not evaluate 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 rhs is taken. All other checks are standard, as well as the auxiliary function *mtype* (defined analogously to *mbody*) 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 converts them to 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[2] 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 condition $U_1\Downarrow \leq U_1'$ 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 a few examples of typings. We start with the following simple class declaration (assuming that class `C` is defined in the same program):

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

```
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 $\texttt{C}^-$ **with** $\texttt{C}^-$ (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 $-;\texttt{this}{:}\texttt{H}^+ \vdash \texttt{new C(this.m())} : \texttt{C}^+?$, therefore we can derive the return type $\texttt{C}^+$ **with** $\texttt{C}^-$ (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 3.1. *If* $\emptyset \vdash e : T$*, and* $e, \emptyset, \emptyset \Downarrow r$*, then* $r \neq \mathtt{w}$.

4. 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. [16, 18, 17] 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 [18]. Coinductive logic programming has proved to be useful for formal verification [13, 15], static analysis and symbolic evaluation of programs [4, 3, 5].

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 [20, 10], 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 [19, 9, 14].

In comparison with the more foundational studies [1, 2] 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, techniques to prove the correctness of corecursive methods could be explored, possibly integrated with proof assistants, as Coq [8], 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 3, 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.

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

[3] 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.

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

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

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

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

[8] 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.

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

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

[11] 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.

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

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

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

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

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

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

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

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