Translating Corecursive Featherweight Java in Coinductive Logic Programming*

Davide Ancona and Elena Zucca

DIBRIS, Università di Genova, Italy
{Davide.Ancona,Elena.Zucca}@unige.it

Abstract. Corecursive FeatherWeight Java (COFJ) is a recently proposed extension of the calculus FeatherWeight Java (FJ), supporting cyclic objects and regular recursion, and explicitly designed to promote a novel programming paradigm inspired by coinductive Logic Programming (coLP), based on coinductive, rather than inductive, interpretation of recursive function definitions.
We present a slightly modified version of COFJ where the application of a coinductive hypothesis can trigger the evaluation of a specific expression at declaration, rather than at use site. Following an approach inspired by abstract compilation, we then show how COFJ can be directly translated into coLP, when coinductive SLD is extended with a similar feature for explicitly solving a goal when a coinductive hypothesis is applied.
Such a translation is quite compact and, besides showing the direct relation between COFJ and coinductive Prolog, provides a first prototypical but simple and effective implementation of COFJ.

1 Introduction

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

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

The COFJ calculus [7] is an extension of FJ where it is possible to define cyclic objects, and where an abstraction mechanism, called regular corecursion, is provided for supporting implementation of coinductive operations on cyclic objects. Its semantics is inspired by the recent results concerning the operational semantics of coinductive Prolog [9,11,10] and the implementation of regular corecursion on top of the standard interpreter based on the inductive semantics of the language [1].

In comparison with the standard semantics of coinductive Prolog, COFJ has a peculiar feature that allows programmers to execute specific code when a coinductive hypothesis is applied; such a characteristic is useful to avoid boilerplate code for all those definitions where the simplest coinductive approach would fail to be correct: a

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

typical example consists in the definition of a `member` function working correctly on cyclic lists.

While in previous work [7] such a feature requires programmers to specify at use site (that is, at method invocation) the expression that will be finally executed when a coinductive hypothesis is applied, here we propose the dual approach where the code is specified at declaration site. Such an alternative design choice has the advantage of making code simpler, without compromising the expressive power of the feature; we have found several cases where the declaration site alternative turns out to be simpler, whereas we were not able to devise examples where the use site approach turns out to be more practical.

A very similar solution has been proposed and experimented with a prototype Prolog meta-interpreter for coinductive Prolog by one of the authors of this paper (see the extended version of the published paper [1]): programmers can define special **`finally`** clauses that are considered when a coinductive hypothesis for a specific predicate is applied.

In this way, coFJ can be translated into coinductive Prolog (with **`finally`** clauses) in a rather direct way, by exploiting the approach of abstract compilation [6,5,2,4,3], a novel approach which aims to reconcile type analysis and symbolic execution, where programs are compiled into a constraint logic program (CLP), and type analysis corresponds to solving a certain goal w.r.t. the coinductive semantics of CLP. In a nutshell, the approach consists in translating a coFJ program into a logic program and a goal, where the goal is obtained by translating the main expression of the program; then, the execution of the program is equivalent to the coSLD resolution of the goal extended with **`finally`** clauses.

The paper is organized as follows: Section 2 presents coFJ, Section 3 shows the semantics of coinductive Prolog with **`finally`** clauses, whereas Section 4 formalizes the translation of coFJ.

2 coFJ

The coFJ calculus 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, and in the formal semantics all methods are implicitly assumed to be all corecursive.

The semantics of coFJ 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 call $v.m(\overline{v})$ corresponds to a previous call which is still active on the stack, then we say that a coinductive hypothesis is applied and such a call terminates immediately, by returning the same value that will be returned by the corresponding call found on the stack.
- method declarations have shape $C\ m(\overline{C\ x})\ \{e$ `with` $e'\}$, where the `with` clause is introduced to provide a value e' returned in place of the standard value e, when

a coinductive hypothesis is applied (see the examples below that motivate its introduction). The standard behavior of regular corecursion is obtained when e' is the special variable `res` denoting the value that would be returned by corecursion; hence, in the sequel the method body $\{e\}$ is just a shortcut for $\{e$ `with res`$\}$.

As a first example, let us consider the following class declarations:

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

FJ does not allow creation of cyclic objects. For instance, though well-typed, any invocation of method `infOcc` does not terminate.

In mainstream object-oriented languages cyclic objects can be constructed and manipulated, but in a very primitive way; to create a cyclic list, field `nx` is initialized with a default value (typically **`null`**), then a finite list is constructed, and, finally, a cycle is introduced by reassigning the proper reference to the field. Such a solution however cannot be adopted by FJ, where assignment is not supported; in mainstream object-oriented languages as Java, it is still possible to create instances of `CycList` even when both fields are **`final`**, but a considerable amount of boilerplate code could be required.[1]

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:

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

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

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

Let us now try to define a method `allPos`, able to work correctly for both non cyclic and cyclic lists, that returns true iff all the elements of the list on which it is invoked are positive.

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

If the list is not cyclic, then the behavior of the method is as expected in the standard inductive case, since the same recursive call cannot occur more than once.

[1] Consider, for instance, the problem of defining a constructor that takes a non cyclic list and builds its corresponding cyclic version.

If the list is cyclic, but contains non positive elements, then the method invocation returns the value **false** and no coinductive hypothesis is applied.

The only case where the coinductive hypothesis is applied occurs when the method is invoked on a cyclic list with all positive elements; however, in this case the return result is not correct, since it corresponds to the undetermined value `X = X`.

To overcome this problem, **true** in place of `res` is returned when a coinductive hypothesis is applied (recall that method body $\{e\}$ is simply an abbreviation for $\{e$ `with res`$\}$).

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

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.

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}
}
```

From the examples above, one could conclude that, when a coinductive hypothesis is applied, a value different from `res` has to be returned only in case of primitive types as **bool**, that is, when recursive equations are not contractive, that is, are not guarded by an instance creation expression, and, hence, the existence of a unique solution is not guaranteed. To show that this is not the case, we define the method `noRep` which, invoked on a possibly 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 obvious semantics. Note that, in case noRep is invoked on the cyclic list L = **new** NEList(0,L), the invocation this.nx.noRep() would be on exactly the same list, hence, 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).

Finally, we end this section with two more examples: the former defines the method isCyc, that returns **true** iff the list on which it is invoked is cyclic, to show an example where the value returned 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. Let us consider the case for non empty lists:

```
    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))).wrongRemPos() 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 retuned. 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 at inspecting the elements of the cyclic part of the list is bound to fail, since the cycle may begin at any arbitrary position in the list. For instance, if l=**new** `NEList(-1,L=`**new** `NEList(1,L))`, then l`.allPos()` evaluates to **false**, while l`.nx.allPos()` evaluates to **true**; therefore, as expected, l`.remPos()` returns **new** `NEList(-1,`**new** `EList()).`

The syntax of coFJ is given in Figure 1. We follow the usual FJ notations and conventions, notably, 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 denoted by $\#\overline{x}$, whereas dom and img represent the domain and image of a map, respectively.

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

Fig. 1. coFJ syntax

Every class has an implicit canonical constructor as in FJ, and we assume the standard FJ well-formedness conditions on a program p. That is: 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 field names declared in a class are distinct from those declared in its superclasses (no field hiding). Furthermore, parameters must be distinct from `this` and `res`. Finally, for every class name C (except `Object`) occurring in p, we have $C \in dom(p)$.

There are the following differences w.r.t. FJ: in addition to the basic sets mentioned above, we assume an infinite set of *labels* X, we omit cast expressions for brevity, method bodies have shape $\{e\ \texttt{with}\ e'\}$, and the definition of values is more general.

Indeed, in FJ values and objects (instances of classes) coincide, and have shape `new` $C(\overline{v})$, that is, are (a concrete representation of) inductive terms, built by constructor invocations. Here, values are generalized, that is, they can be annotated with labels, and a (sub)value can be a (reference to a) label, expected to annotate an enclosing

value. Objects are values which are not labels, that is, of form X_1=... X_n=new $C(\overline{v})$, abbreviated $\overline{X =}$ new $C(\overline{v})$ with our convention.

We say that a value is *closed* if all references contained in it are bound to existing labels. Closed values are a concrete representation of either regular terms built by constructor invocations, or the undetermined value. Note that the only closed values which are not objects have shape X_1=... X_n=X_i, with $i \in 1..n$, and are all equivalent representations of the undetermined value. 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.

Values which are not objects (notably, representations of the undetermined value and labels) can be safely passed as arguments and obtained as result of field access and method invocation, but the semantics is undefined when they are used as receivers in field accesses and method invocations.

The big-step semantics $e, \sigma, \pi \Downarrow r$ returns the result r, if any, of evaluating an expression e in the context of an invocation stack σ keeping track of pending method invocations, and of a frame π defining the values of all local variables (that is, all formal parameters, and the special variables `this` and `res`). This relation should be indexed over programs, however for brevity we leave implicit such an index in all judgments defined in the paper. Formally, σ is a finite and injective map from expressions of the form $v.m(\overline{v})$, which we call *(invocation) redexes*, to labels X, whereas π is, as usual, a finite map from variables to values.

Rules are given in Figure 2. For lack of space, we omit standard technical details, notably, the formal definitions of stack update $\sigma[v.m(\overline{v}){:}X]$, and of parallel substitution $v_1[v_2/\overline{X}]$, and the auxiliary functions *fields*, *mbody*. We write $\overline{e}, \sigma, \pi \Downarrow \overline{v}$ as a shorthand for the set of judgments $e_1, \sigma, \pi \Downarrow v_1 \ldots e_n, \sigma, \pi \Downarrow v_n$.

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

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

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

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

Fig. 2. coFJ big-step rules

Rule (VAR) is straightforward.

Rule (FIELD) models field access. Recall that, with the FJ convention, $\overline{C\ f\texttt{;}}$ stands for $C_1\ f_1\texttt{;}\ \ldots\ C_n\ f_n\texttt{;}$. 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 the class

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

the field access $v.\texttt{f}$, with

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

is reduced to

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

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

If a method invocation does not correspond to any redex on the invocation stack, 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 method body in the new frame where `this` and the formal parameters are associated with the receiver object and the arguments, respectively. The evaluation of the method body is performed by keeping track of the new redex (formally, adding in σ an association from this redex to a fresh label *X*). In this way, method invocations leading to the same redex will be no longer handled by this rule (hence avoiding non termination). Moreover, when the evaluation of the method body is completed, references to *X* in the resulting value (due to recursive method invocations leading to the same redex handled by rule (COREC)) are bound. This mechanism makes it possible to obtain a cyclic object as the result of a method invocation.

Rule (COREC) handles the case when a coinductive hypothesis is applied, hence, the invocation redex is found on the invocation stack σ; the expression e' specified after **`with`** is evaluated in the new frame where `this` is associated with the receiver object, `res` is associated with the label *X* found on the invocation stack, and the formal parameters are associated with the arguments.

Finally, (NEW) is the standard rule for constructor invocation. The side condition ensures that the constructor is invoked with the appropriate number of arguments.

3 Coinductive Prolog with `finally` clauses

It is known [9,1] that properties which are existentially quantified on cyclic data (as membership for regular lists) cannot be defined easily in a coinductive way, and, hence, rather involved and ad hoc solutions have to be devised.

In recent work ([1], see the extended version) we have proposed a new feature aiming to solve this problem, by allowing the user to define the specific behavior of a coinductive predicate when an atom is solved by coinductive hypothesis, by means of **finally** clauses.

While facts are used in Prolog for defining the base cases for induction, **finally** clauses specify the behavior in case of application of the coinductive hypothesis in regular coinduction.

Let us first introduce **finally** clauses with the definition of predicate `member`.

```
:- use_module(cosldmeta2finally).

coinductive(member(_,_)).

member(N,[N|_]).
member(N,[_|L]) :- member(N,L).
finally(member(_,_)) :- fail.
```

In the case of `member` the coinductive hypothesis is applied when all the elements of the cyclic list has been already inspected; this means that none of them was found equal to the first argument, therefore in this case the goal must fail. The last clause with **finally** is used for specifying such a behavior: when a coinductive hypothesis can be applied for `member` (independently of the arguments), then the goal must fail.

We define the semantics of **finally** clauses with a Prolog meta-interpreter; while meta-interpreters are not efficient to be suitable for practical uses, the meta-programming facilities offered by Prolog are an ideal tool to experiment with new semantics and programming abstractions: Prolog meta-interpreters are concise and abstract enough to serve as a formal semantics, yet they provide prototype implementations to test new language features.

The meta-interpreter defining the coinductive semantics of a logic program extended with **finally** clauses is given by the following Prolog program:

```
:- use_module(library(ordsets)).

cosld(G) :- ord_empty(E),solve(E,G).
solve(H, (G1,G2)) :- !,solve(H, G1), solve(H,G2).
solve(_,A) :- inductive(A),!,A.
solve(H,A):- found(A, H),(clause(finally(A),As) *-> solve(H,As);
   true).
solve(H,A):- !,clause(A,As),insert(H,A,NewH),solve(NewH,As).

inductive(A) :- predicate_property(A,built_in),!.
inductive(A) :- predicate_property(A,file(AbsPath)),
             file_name_on_path(AbsPath,library(_)),!.
inductive(A) :- !,\+ coinductive(A).

insert(L1,A,L2) :- is_in(A,L1) -> fail;ord_add_element(L1,A,L2).

is_in(A1,[A2|_]) :- unifiable(A1,A2,_),!.
is_in(A,[_|L]) :- is_in(A,L),!.
```

```
found(A,H) :- memberchk(A,H).
```

The meta-interpreter is a variation of the standard operational semantics of coinductive Prolog.

As usual, if an atom is inductive, then it is directly solved by the Prolog interpreter; the cut allows the meta-interpreter to skip the clauses dealing with coinduction. Predicates are inductive by default, those coinductive (and necessarily user-defined) have to be explicitly specified by the user. Therefore the inductive predicates are either built-in or imported from a standard library or they have not been declared coinductive.

This solution enforces a stratification between coinductive and inductive predicates: while a coinductive predicate can be defined in terms of an inductive one, the opposite is not allowed; this restriction avoids contradictions due to naive mixing of coinduction and induction [10].

Besides **`finally`** clauses, the meta-interpreter departs from the standard operational semantics of coinductive Prolog also because if performs a pruning of the search tree to avoid some kinds of non terminating failures. A pruning of the search trees can be performed by applying a clause only if the atom to be solved does not unify with a coinductive hypothesis, after it has been unified with the head of the clause. Predicate `insert` fails if the atom to be inserted in the list of coinductive hypotheses unifies with some coinductive hypothesis already contained in the list.

Let us show now how **`finally`** clauses affect the semantics of a logic program. For keeping the treatment simple, **`finally`** is managed as a predicate symbol. **`finally`** clauses are managed by the third clause for `solve/2`, which deals with the application of the coinductive hypothesis. If the current atom `A` unifies with some coinductive hypothesis in `H` (that is, `found(A,H)` succeeds), then the meta-interpreter checks whether there exists a **`finally`** clause applicable for `A`, with the atom `clause(`**`finally`**`(A),As)`; if it is the case, then the body of the corresponding **`finally`** clause is solved; if no **`finally`** clause is found, then the default behavior is implemented: the atom `A` succeeds.

The built-in predicate `*->` has been used instead of `->`, to allow backtracking for the resolution of `clause(`**`finally`**`(A),As)`.

4 Encoding of coFJ

The translation of coFJ is inspired by the abstract compilation approach [6,5,2,4,3], where type analysis of an object-oriented program is performed by translating it in a suitable logic program, and by solving a goal (corresponding to the translation of the main expression) according to the coinductive interpretation of the logic program.

In this case the compilation is not abstract, but faithfully respect the operational semantics of coFJ.

We first explain the meaning of all predicates used in the translation.

- $extends(C, C')$: class C extends class C';
- $dec\_field(C, f)$: class C declares field f;
- $dec\_fields(C, [f_1, \ldots, f_n])$: $f_1, \ldots, f_n$ are all fields declared in class C;

- $dec\_meth(C, m)$: class C declares method m;
- $merge(l_f, l_v, l_{(f,v)}, l'_v)$: the list of fields l_f is merged with the list of values l_v to constitute the list of pairs (field,value) $l_{(f,v)}$. Lists are processed from left to right, and l_v is allowed to contain more elements than l_f; l'_v contains the list of values that have not been associated with a corresponding field in l_f;
- $new(C, [v_1, \ldots, v_n], v)$: invoking the constructor of C with arguments $v_1, \ldots, v_n$ returns the value v;
- $field\_acc(v_1, f, v_2)$: accessing field f of v_1 yields value v_2;
- $invoke(v_0, m, [v_1, \ldots, v_n], v)$: invoking method m on v_0 with arguments $v_1, \ldots, v_n$ returns the value v;
- $has\_meth(C, m, [v_0, \ldots, v_n], v)$: if no corecursive hypothesis is applicable, then invoking method m looked up from class C with target object v_0 and arguments $v_1, \ldots, v_n$ returns the value v;
- $with\_meth(C, m, [v_0, \ldots, v_n], v)$: if a corecursive hypothesis is applicable, then invoking method m looked up from class C with target object v_0 and arguments $v_1, \ldots, v_n$ returns the value v.

Figure 3 defines the syntax-directed translation of a COFJ program into a corresponding logic program. A part of the generated Horn clauses, denoted by P_{tr}, are independent from the input program, and, hence, are specified separately (see Figure 4).

- $tr(\overline{cd}) = P_{tr} \bigcup_{cd \in \overline{cd}} tr(cd)$
- $tr(\texttt{class}\ C\ \texttt{extends}\ C'\ \{\ \overline{fd}\ \overline{md}\ \}) =$
  $\{extends(C, C')\} \bigcup_{fd \in \overline{fd}} tr(C, fd) \bigcup_{md \in \overline{md}} tr(C, md)$
- $tr(C, C'\ f\texttt{;}) = \{dec\_field(C, f)\}$
- $tr(C, C_0\ m(\overline{C\ x})\ \{e\ \texttt{with}\ e'\}) =$
  $$\left\{ \begin{array}{l} dec\_meth(C, m) \\ has\_meth(C, m, [This, var(\overline{x})], V) \leftarrow B \\ \\ with\_meth(C, m, [This, var(\overline{x})], Res) \leftarrow B', Res = V' \end{array} \right\}$$
  if $tr(e) = (B, V)$ and $tr(e') = (B', V')$
- $tr(x) = (true, var(x))$
- $tr(e.f) = ((B, field\_acc(V, f, V')), V')$ if $tr(e) = (B, V)$, and V' fresh logical variable
- $tr(e_0.m(\overline{e})) = ((B_0, \overline{B}, invoke(V_0, m, [V_1, \ldots, V_n], V)), V)$
  if $tr(e_0) = (B_0, V_0)$, $tr(\overline{e}) = (\overline{B}, \overline{V})$, and V fresh logical variable
- $tr(\texttt{new}\ C(\overline{e})) = ((\overline{B}, new(C, [V_1, \ldots, V_n])), V)$
  if $tr(\overline{e}) = (\overline{B}, \overline{V})$, and V fresh logical variable

Fig. 3. Translation of COFJ

The function tr is overloaded and applied to different syntactic categories; for field and method declarations the function takes a first additional argument corresponding to the class where the declaration is contained. For class, method and field declarations, the function returns a set of clauses, whereas for expressions it returns a pair consisting of a conjunction of atoms B, and a logical variable V, where V occurs in B (if B is

not empty) and represents the value returned by the evaluation of the expression. The empty conjunction of atoms is denoted by $true$.

For simplicity we assume that class, field and method names need not to be translated (this assumption is reasonable if the standard Java convention is followed for field and method names, whereas for class names the first letter of class names should be turned into the corresponding lowercase), whereas we assume the existence of a suitable auxiliary injective function var mapping COFJ variables to logical variables (for instance, by capitalizing them) such that $var(\texttt{this})$=*This*, and $var(\texttt{res})$=*Res*.

The translation of a sequence of class declarations $\overline{cd}$ returns all clauses obtained by translating each single class declaration, together with the clauses P_{tr} which do not depend on a particular input COFJ (see Figure 4 below).

A class declaration $\texttt{class}\ C\ \texttt{extends}\ C'\ \{\ \overline{fd}\ \overline{md}\ \}$ is translated into a set of clauses consisting of the fact recording that C is a class declared in the program and extends C', and of the clauses obtained by translating all field and method declarations in $\overline{fd}$ and $\overline{md}$ w.r.t. C.

Translating a declaration of field f in class C simply produces a fact specifying that f is declared in C.

The translation of a method declaration $C_0\ m(\overline{C\ x})\ \{e\ \texttt{with}\ e'\}$ in class C produces three different clauses. The first is a fact that simply states that *m* is declared in C.

The second clause $has\_meth(C, m, [This, var(\overline{x})], V) \leftarrow B$ specifies the behavior of the method when no coinductive hypothesis is applied: when invoked on object $This$ and arguments $var(\overline{x})$, method *m* of class C returns the value V; the body B and the logical variable V are obtained by translating the expression *e* in the body of the method.

The last clause $with\_meth(C, m, [This, var(\overline{x})], Res) \leftarrow B', Res = V'$ specifies the behavior of the method when a coinductive hypothesis is applied: in this case, if the receiver and the arguments are $This$ and $var(\overline{x})$, respectively, then method *m* of class C returns the result Res that equals the logical variable V' corresponding to the returned value of expression e'; the body B' and the logical variable V' are obtained by translating the expression e'. Recall that we assume that the special variable $\texttt{res}$ is translated to Res by var.

The translation of a variable is straightforward.

Field access $e.f$ is translated to the pair $((B, field\_acc(V, f, V')), V')$, where the conjunction of atoms B and the logical variable V are obtained by translating the expression *e*; the atom $field\_acc(V, f, V')$ states that the value of field f of V is V', where V' is a fresh variable not occurring in B, corresponding to the value of $e.f$.

The pair $((B_0, \overline{B}, invoke(V_0, m, [V_1, \ldots, V_n], V)), V)$ is returned by the translation of expression $e_0.m(\overline{e})$; the conjunctions of atoms B_0 and $\overline{B}$, and the logical variables V_0 and $\overline{V}$ are obtained by translating the expressions e_0 and $\overline{e}$, respectively. The atom $invoke(V_0, m, [V_1, \ldots, V_n], V))$ states that, when invoked on object V_0 and arguments $\overline{V}$, method *m* returns value V, where V is a fresh variable not occurring in B_0 and $\overline{B}$, corresponding to the value of $e_0.m(\overline{e})$.

Expression $\texttt{new}\ C(\overline{e})$ is translated to the pair $((\overline{B}, new(C, [V_1, \ldots, V_n]), V)$, where the conjunctions of atoms $\overline{B}$, and the logical variables $\overline{V}$ are obtained by translating the expressions $\overline{e}$. The atom $((\overline{B}, new(C, [V_1, \ldots, V_n]), V)$ states that invocation of the

constructor of C with arguments $\overline{V}$ returns value V, where V is a fresh variable not occurring in $\overline{B}$, corresponding to the value of `new` $C(\overline{e})$.

Figure 4 contains the clauses P_{tr}, emitted by the translation, that do not depend on a particular input.

$$
\begin{array}{l}
coinductive(invoke(\_,\_,\_,\_)).\\
coinductive(has\_meth(\_,\_,\_,\_)).\\
coinductive(with\_meth(\_,\_,\_,\_)).\\
\\
dec\_fields(C, L) \leftarrow setof(F, dec\_field(C, F), L), !.\\
dec\_fields(\_, [\,]).\\
\\
new(object, [\,], obj(object, [\,])).\\
new(C, Args, obj(C, FVL)) \leftarrow dec\_fields(C, FL), merge(FL, Args, CFVL, PArgs),\\
\qquad extends(C, P), new(P, PArgs, obj(P, PFVL)), append(CFVL, PFVL, FVL).\\
\\
field\_acc(obj(\_, FVL), F, V) \leftarrow member((F, V), FVL).\\
\\
invoke(TO, M, Args, RV) \leftarrow TO = obj(C, \_), has\_meth(C, M, [TO|Args], RV).\\
finally(invoke(TO, M, Args, RV)) \leftarrow TO = obj(C, \_), with\_meth(C, M, [TO|Args], RV).\\
\\
has\_meth(C, M, Args, RV) \leftarrow\\
\quad \neg dec\_meth(C, M), extends(C, P), has\_meth(P, M, Args, RV).\\
\\
with\_meth(C, M, Args, RV) \leftarrow\\
\quad \neg dec\_meth(C, M), extends(C, P), with\_meth(P, M, Args, RV).
\end{array}
$$

Fig. 4. Definition of P_{tr}

The only predicates that are required to be corecursive are $invoke$, has_meth and $with\_meth$ dealing with method invocation.

An object value is represented by the term $obj(C, [(f_1, v_1), \ldots, (f_n, v_n)])$, where C is the class of the object, whereas $[(f_1, v_1), \ldots, (f_n, v_n)]$ is the list of pairs associating a value with each field of the object.

Predicates $member$ and $append$ are the standard library predicates on lists.

The clause for $invoke$ defines the behavior of method invocation when no coinductive hypothesis is applied; the class C of the target object TO is retrieved, and then the returned value is that specified by predicate has_meth.

The **`finally`** clause for $invoke$ defines the behavior of method invocation when a coinductive hypothesis is applied; analogously to the previous case, the class C of the target object TO is retrieved, and then the returned value is that specified by predicate $with\_meth$.

The two clauses for has_meth and $with\_meth$ propagate method look up to the parent class, when the searched method is not found in the current class; both clauses use a simple form of negation, since predicate dec_meth is defined by a collection of ground atoms.

5 Conclusive remarks

Corecursive definitions can become simpler if programmers are allowed to specify through an expression the value that has to be returned in case a coinductive hypothesis is applied. The usefulness of such a feature has been investigated both in coinductive Prolog [1] (by means of **`finally`** clauses), and in CoFJ [7].

In this paper we exploit the approach of abstract compilation to translate CoFJ in coinductive Prolog with **`finally`** clauses. The defined translation is quite compact and, besides showing the direct relation between CoFJ and coinductive Prolog, provides a first prototypical but simple and effective implementation of CoFJ; by employing an experimental implementation of coinductive Prolog with **`finally`** clauses, by means of a Prolog meta-interpreter, the semantics of CoFJ can be implemented almost for free with a syntax directed translation.

Regular corecursion versus lazy evaluation

Corecursion [8] has been used in some contexts to denote the ability, supported by lazy evaluation, of defining a function that produces some infinite data in terms of the function and the data itself.

As an example, let us consider the following Haskell code defining the infinite stream $!0 : 1! : 2! : \ldots$ of the factorials of all natural numbers.

```
fact_stream = 1:gen_fact 1 1
gen_fact  n m = let k = n*m in k:gen_fact k (m+1)
```

The stream `fact_stream` is defined in terms of the corecursive function `gen_fact`.

After having defined `fact_stream`, one can get the factorial of n by simply selecting the element at position n in `fact_stream`:

```
*Main> fact_stream !! 10
3628800
```

Though the stream is infinite, it is possible to access any arbitrary element because the list constructor '`:`' is non-strict and, hence, the call to function `gen_fact` is computed lazily. More abstractly, the data returned by `gen_fact` corresponds to a tree whose depth is infinite, and that is **not** regular.

Now let us try to check whether all elements in the stream are greater than 0, with the predefined function `all`.

```
*Main> all (\x -> x>0) fact_stream
-- does not terminate
```

Checking that an arbitrary predicate holds on all the factorials of natural numbers is only semi-decidable: termination is guaranteed only if the predicate does not hold for some element, as in `all (\x -> x<100) fact_stream`.

Let us now consider this other stream declaration:

```
ones = 1:ones
```

Differently from `fact_stream`, stream `ones` is regular. Such a stream is defined as a cyclic data structure, and no lazy evaluation is required: it is recursively defined by using just the list constructor.

Despite the regularity of `ones`, the evaluation of `all (\x -> x>0) ones` does not terminate in Haskell, because the logical conjunction `&&` is strict in its second argument (when the first argument evaluates to `True`), and `all` is defined in the standard inductive way.

This problem does not occur in CoFJ, as shown in Section 2 with method `allPos`. Adopting a lazy evaluation strategy for supporting programming with cyclic objects would be a too radical shift in the semantics of object-oriented languages. Indeed, cyclic objects are just regular values, that is, a very particular case of infinite objects.

References

1. 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`.
2. 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 *Post-proceedings of Formal Verification of Object-Oriented Software (FoVeOOS 2010)*, volume 6528, 2011. Selected paper.
3. D. Ancona and G. Lagorio. Coinductive type systems for object-oriented languages. In *ECOOP'09 - Object-Oriented Programming*, volume 5653, pages 2–26, 2009. Best paper prize.
4. D. Ancona and G. Lagorio. Coinductive subtyping for abstract compilation of object-oriented languages into Horn formulas. In *Proceedings of GandALF 2010*, volume 25, pages 214–223, 2010.
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 G. Lagorio. Static single information form for abstract compilation. In *IFIP Theoretical Computer Science 2012*, 2012. To appear.
7. D. Ancona and E. Zucca. Corecursive Featherweight Java. In *FTfJP '12*, 2012. An extended version submitted for publication is available at `ftp://ftp.disi.unige.it/person/AnconaD/AnconaZucca12.pdf`.
8. J. Barwise and L. Moss. Vicious circles: On the mathematics of non-wellfounded phnenomena. *J. of Logic, Lang. and Inf.*, 6:460–464, 1997.
9. L. Simon. *Extending logic programming with coinduction*. PhD thesis, University of Texas at Dallas, 2006.
10. L. Simon, A. Bansal, A. Mallya, and G. Gupta. Co-logic programming: Extending logic programming with coinduction. In *ICALP 2007*, pages 472–483, 2007.
11. L. Simon, A. Mallya, A. Bansal, and G. Gupta. Coinductive logic programming. In *ICLP 2006*, pages 330–345, 2006.