Fundamenta Informaticae FICS'13 Special Issue 1–29
DOI 10.3233/FI-2012-0000
IOS Press

CoCaml: Functional Programming with Regular Coinductive Types

Jean-Baptiste Jeannin
Department of Computer Science
Carnegie Mellon University
jeannin@cs.cmu.edu

Dexter Kozen
Department of Computer Science
Cornell University
kozen@cs.cornell.edu

Alexandra Silva [C]
Department of Computer Science
Radboud University Nijmegen
alexandra@cs.ru.nl

Abstract. Functional languages offer a high level of abstraction which results in elegant, easy to understand, programs. Central to the development of functional programming are inductive and coinductive types and associated programming constructs, such as pattern-matching. Whereas inductive types have a long tradition and are well supported in most languages, coinductive types are subject of more recent research and less mainstream.

We present CoCaml, a functional programming language extending OCaml, which allows us to define recursive functions on regular coinductive datatypes. These functions are defined like usual recursive functions, but parameterized by an equation solver. We present a full implementation of all the constructs and solvers and show how these can be used in a variety of examples, including operations on infinite lists, infinitary λ-terms and p-adic numbers.

keywords: regular coinductive types, functional programming, recursion

Keywords: Context free languages, Kleene algebra, algebraically complete semirings, Conway semirings, μ-semiring

1. Introduction

Functional languages offer elegant constructs to manipulate datatypes and define functions on them. Their inherent high level of abstraction, which avoids explicit manipulation of pointers or references, has

[C]Corresponding author

played a key role in the increasing popularity of languages such as Haskell and OCaml. The combination of pattern-matching and recursion provides a powerful tool for computing with algebraic datatypes such as finite lists or trees. However, for coinductive objects such as infinite lists or trees, the situation is less clear-cut. The foundations of coinductive types are much younger and hence all the theoretical developments have not yet fully found their place in mainstream implementations.

Despite infinite objects being hard to tackle in general, several classes of them are more manageable. For instance, ultimately periodic infinite lists or infinite trees with finitely many subtrees should offer fewer challenges than lists/trees with no apparent regular structure. This class of coinductively defined objects has been widely studied in the literature under the name of regular/rational types.

One might wonder whether the restriction to regular coinductive datatypes might make computing with coinductive datatypes less attractive. On the contrary, regular coinductive types have a wide range of applications and occur in many areas of computer science. For instance, unfoldings of finite graphs and automata are prime examples of regular coinductive types. They can also be found in formal grammar and data flow analysis in compiler construction. From a theoretical perspective, they also offer many challenges in devising sound definition and proof principles.

This paper presents a full implementation of language constructs in OCaml for manipulating regular coinductive datatypes. One can already define them in OCaml, but the means to define functions on them are limited. Often the obvious definitions do not halt or provide the wrong solution. We do not change the way datatypes are defined. Instead, we provide constructs that allow the programmer to specify how to solve equations resulting from usual recursive definitions when these are applied to regular coinductive objects.

Let us provide some motivation using an example of a function over one of the simplest coinductive datatypes: infinite lists. The regular elements of this datatype are the eventually periodic infinite lists. The type of finite and infinite lists of OCaml, `'a list`, is a built-in coinductive type consisting of two cases, `[]` for the empty list and `hd :: tl` for the list with head `hd` of type `'a` and tail `tl` of type `'a list`. Concrete regular infinite lists can then be defined coinductively using the **`let rec`** construct:

```
let rec ones = 1 :: ones
let rec alt  = 1 :: 2 :: alt
```

The first example defines the infinite sequence of ones $1, 1, 1, 1, \ldots$ and the second the alternating sequence $1, 2, 1, 2, \ldots$.

Although the **`let rec`** construct allows us to specify regular infinite lists, further investigation reveals a major shortcoming. For example, suppose we wanted to define a function that, given an infinite list, returns the set of its elements. For the lists `ones` and `alt`, the function should return the sets $\{1\}$ and $\{1, 2\}$, respectively. Note that by regularity, the set of elements is always finite. One would like to write a function definition using equations that pattern-match on the two constructors of the `list` datatype:

```
let rec set l = match l with
   | [] -> []
   | h :: t -> insert h (set t)
```

where `insert` adds an element to a set, represented by a finite list without duplicates. Sadly, this function will not halt in OCaml on `ones` and `alt`, even though it is clear what the answers should be.

This problem arises from the fact that the standard semantics of recursion does not cope well with regular coinductive objects. This has also been observed by others [21, 4, 7, 9, 30]. In [30], the authors discuss the design of language constructs that could avoid the problem above and provide a mock-up

implementation in an OCaml-like language. In this paper, we give a full implementation and improve on those results by showing how an alternative semantics, provided by equation solvers, can be given in a lightweight, elegant fashion.

In this paper, we present CoCaml, an extension of OCaml in which functions defined by recursive equations can be supplied with an extra parameter, namely a solver for the equations generated when the function is applied to an argument which might be a regular coinductive object. The contributions of this paper can be summarized as follows:

1. Implementation of a new language construct `corec[solver]`, which takes an equation solver as an argument.

2. Implementation of several generic solvers, which can be used as arguments of the `corec` construct. These include `iterator`, a solver to compute fixpoints; `constructor`, used to compute regular coinductive objects; `gaussian`, which solves a system using gaussian elimination; `separate`, which splits regular lists into a finite prefix and a cyclic part, enabling easy display of regular infinite lists. We also provide the user with means to define custom solvers using a `Solver` module.

3. A large set of examples illustrating the simplicity and versatility of the new constructs and solvers. These include a library for p-adic numbers and several functions on infinite lists and λ-terms.

The importance of regular infinite objects in functional and coinductive logic programming has been recognized by many authors [47, 19, 21, 44, 5, 4, 3, 40, 14, 45, 7, 9, 8, 48], and research is ongoing. The chief distinguishing characteristic of our work is that we provide powerful programming language tools in the form of a versatile set of equation solvers that allow the programmer to realize the full potential of regular coinductive datatypes. A more detailed comparison with existing work will be made in a later section.

The paper is organized as follows. In §2, we present the proposed solution for the motivational example of the introduction and give a high level overview of the implementation and subtleties to be tackled. In §3, we describe regular coinductive types in the context of a functional language and *capsule semantics*, a heap-free mathematical semantics for higher order functional and imperative programs, which provides the foundation for our implementation. In §4, we describe the implementation of several generic solvers and examples of their applicability. We also provide a module that enables the user to define custom solvers and illustrate its applicability by defining a new solver. In §5, we give several detailed examples illustrating the use of the new constructs and solvers, including functions on potentially infinite lists, p-adic numbers, and infinitary λ-terms. In §6, we dive into the details behind the implementation. We discuss related work in §7 and in §8 we conclude and give directions for future work.

2. Overview

Let us go back to the `set` example from the introduction and explain the steps we took in order to obtain the desired solution in OCaml.

Note that the definition of `set` is not corecursive, as we are not asking for a greatest solution or a unique solution in a final coalgebra, but rather a least solution in a different ordered domain from the one provided by the standard semantics of recursive functions. The standard semantics of recursive functions

gives us the least solution in the flat Scott domain with bottom element $\bot$ representing nontermination, whereas we would like the least solution in a different CPO, namely $(\mathcal{P}(\mathbb{Z}), \subseteq)$ with bottom element $\varnothing$.

When trying to compute the solution of `set(alt)` the compiler generates two equations, namely

```
set(alt) = insert 1 (set(2::alt))
set(2::alt) = insert 2 (set(alt))
```

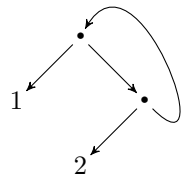
The occurrence of `set(alt)` in the right hand-side of the second equation is what causes the nontermination in the presence of the usual semantics of recursive functions.

The intended solution in this particular case is the least solution of the above equations in the domain $\mathcal{P}(\mathbb{Z})$ ordered by set inclusion. The power of specifying how to solve equations in the codomain is the main feature of our proposed solution.

For instance, the example above would be almost the same in CoCaml:

```
let corec[iterator []] set l = match l with
   | [] -> []
   | h :: t -> insert h (set t)
```

The construct `corec` with the parameter `iterator []` specifies to the compiler that the equations above should be solved using an iterator—in this case a least fixpoint computation—starting with the initial element `[]`. For the infinite list `alt`, which can abstractly be thought of as the circular structure



our extended compiler will generate two equations (nodes are given fresh names),

```
set(x) = insert 1 (set(y))
set(y) = insert 2 (set(x))
```

and then solve them using the specified solver `iterator`, which will produce the intended set $\{1, 2\}$.

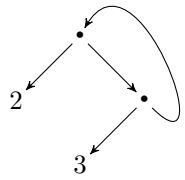
As another example, in which a different type of solver is required, consider the `map` function, which applies a given function to every element of a given list. Again, the obvious definition, when applied to a circular structure, will not halt in OCaml. In CoCaml, we can specify that we want to get a solution with the same structure as the argument. Again, the definition looks very much like the standard one:

```
let corec[constructor] map arg = match arg with
   | f, [] -> []
   | f, h :: t -> f(h) :: map(f,t)
```

As desired, applications of `map` to circular structures halt and produce the expected result. For instance, `map plusOne alt` first generates two equations

```
map(x) = 1 :: (map(y))
map(y) = 2 :: (map(x))
```

Solving the equations produces the infinite list $2, 3, 2, 3, \ldots$ as represented by the circular structure



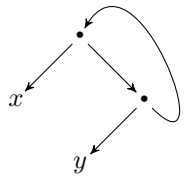
Another motivating example, which we borrow from [30], is the set of free variables of an infinitary λ-term (i.e. a λ-coterm). For ordinary well-founded λ-terms, the following definition works:

```
type term =
   | Var of string
   | App of term * term
   | Lam of string * term

let rec fv = function
   | Var v -> [v]
   | App (t1,t2) -> union (fv t1) (fv t2)
   | Lam (x,t) -> remove x (fv t)
```

However, if we call the function on an regular coterm, say

```
let rec t = App (Var "x", App (Var "y", t))
```



then the function will diverge. However, CoCaml can compute the desired solution $\{x, y\}$ using the `iterator []` solver as in the example above involving elements of an infinite regular list.

Note that regular coinductive objects always have a finite representation in memory, i.e., they are infinite but can be represented finitely with cycles. This is different from a setting in which infinite elements are represented lazily and can be computed on the fly. A few of our examples, like substitution on infinitary λ-terms or mapping a function on an infinite list, could be computed by lazy evaluation, but most of them, such as free variables, cannot. If one implements the free variables example in Haskell then one can not display the resulting set when the function is applied to a regular infinite term. One can still do membership queries, such as *is variable x in `fv t`?* If the answer is positive Haskell will be able to return it, however negative answers will be lost in an infinite search. Also note that even in the cases where computation by lazy evaluation can be done and the regularity of the argument is preserved this cannot be observed: what can then be observed are finite parts of the infinite unfolding of the term.

One should be careful not to confuse computing with regular coinductive objects with computing on graphs or other forms of cyclic data structures. Coinductive objects are not graphs (although they are represented internally as graphs). They are more accurately described as rational or regular elements of a final coalgebra. They may be infinite, but they have a finite representation, as mentioned above. Functions defined on them should be independent of the representation, which means that the function should give equivalent results when applied to equivalent inputs. Here *equivalent* means the finite representation unfolds to the same infinite object, or more accurately, has the same image in the final coalgebra.

The new construct has the big advantage that no change is needed in the specification of the datatype (like in e.g [21]) or in the usual definition of functions. The detection of whether the input is regular coinductive, i.e., it has cycles is done automatically. In this automatic detection, subtleties on how to determine equality of two coinductive elements arise (see §6.3). Another feature of our implementation is the flexibility in the choice of the desired solver. We cast under the same umbrella different solution methods. We provide several generic and versatile solvers (see §4), of which `iterator` and `constructor` are examples, and we provide the user with the means to easily define his/her own solvers (see §4.6).

3. Preliminaries

In this section, we present the basics of coinductive types and the theoretical foundations on well-definedness of functions on coinductive types, which we will use to define the new language constructs. We also describe *capsule semantics*, a heap-free mathematical semantics for higher order functional and imperative programs, on which our implementation is based.

3.1. ML with Coalgebraic Datatypes

Coalgebraic (coinductive) datatypes are very much like algebraic (inductive) datatypes in that they are defined by recursive type equations. The set of algebraic objects form the least (initial) solution of these equations and the set of coalgebraic objects the greatest (final) solution.

Algebraic types have a long history going back to the initial algebra semantics of Goguen and Thatcher [23]. They are very well known and are heavily used in modern applications, especially in the ML family of languages. Coalgebraic types, on the other hand, are the subject of more recent research and are less well known. Not all modern functional languages support them—for example, Standard ML and F# do not—and even those that do support them do not do so adequately.

The most important distinction is that coalgebraic objects can have infinite paths, whereas algebraic objects are always well-founded. *Regular* coalgebraic objects are those with finite (but possibly cyclic) representations. We would like to define recursive functions on coalgebraic objects in the same way that we define recursive functions on algebraic data objects, by structural recursion. However, whereas functions so defined on well-founded data always terminate and yield a value under the standard semantics of recursion, this is not so with coalgebraic data because of the circularities.

In Standard ML, constructors are interpreted as functions, and thus coinductive objects cannot be formed. Whereas in OCaml, coinductive objects can be defined, and constructors are not functions. Formally, in call-by-value languages, constructors can be interpreted as functions under the algebraic interpretation, as they are in Standard ML, but not under the coalgebraic interpretation as in OCaml. In Standard ML, a constructor is a function:

```
- SOME;
val it = fn : 'a -> 'a option
```

Since it is call-by-value, its arguments are evaluated, which precludes the formation of coinductive objects. In OCaml, a constructor is not a function. To use it as a function, one must wrap it in a lambda:

```
> Some;;
Error: The constructor Some expects 1 argument(s),
```

```
       but is applied here to 0 argument(s)
> fun x -> Some x;;
- : 'a -> 'a option = <fun>
```

This allows the formation of coinductive objects:

```
> type t = C of t;;
type t = C of t
> let rec x = C x;;
val x : t = C (C (C (C (C (C (C ...)))))))
```

Despite these differences, inductive and coinductive data share some strong similarities. We have mentioned that they satisfy the same recursive type equations. Because of this, we would like to define functions on them in the same way, using constructors and destructors and writing recursive definitions using pattern matching. However, to do this, it is necessary to circumvent the standard semantics of recursion, which does not necessarily halt on cyclic objects. It has been argued in [30] that this is not only useful, but feasible. In [30], new programming language features that would allow the specification of alternative solutions and methods to compute them were proposed, and a mock-up implementation was given that demonstrated that this approach is feasible. In this paper, we take this a step further and provide a full implementation in an OCaml-like language. We also give several new examples of its usefulness in addition to the examples of [30]. More importantly, we implement solvers in a much more simplified and elegant manner and we provide the user with the ability of specifying his/her own solvers without having to do so directly in the interpreter.

For full functionality in working with coalgebraic data, *mutable variables* are essential. Current functional languages in the ML family do not support mutable variables; thus true coalgebraic data can only be constructed explicitly using **`let rec`**, provided we already know what they look like at compile time. Once constructed, they cannot be changed, and they cannot be created dynamically. This constitutes a severe restriction on the use of coalgebraic datatypes. One workaround is to simulate mutable variables with references, but this is ugly; it corrupts the algebraic typing and forces the programmer to work at a lower pointer-based level. Capsules, which we describe next, offer the right abstraction to avoid the use of reference, making construction and manipulation of coalgebraic data easy.

3.2. Capsule Semantics

Our implementation is based on *capsule semantics* [29], a heap-free mathematical semantics for higher order functional and imperative programs. In its simplest form, a *capsule* is a pair $\langle e, \sigma \rangle$, where e is a λ-term and σ is a partial map with finite domain from variables to λ-terms such that

- $FV(e) \subseteq \operatorname{dom} \sigma$, and
- for all $x \in \operatorname{dom} \sigma$, $FV(\sigma(x)) \subseteq \operatorname{dom} \sigma$

where $FV(e)$ denotes the set of free variables of e. (In practice, a capsule also contains local typing information, which we have suppressed here for simplicity.) Capsules are essentially finite coalgebras; more precisely, a capsule is a finite coalgebraic representation of a regular closed λ-coterm.

In capsule semantics, regular coinductive types and recursive functions are defined in the same way. There is a special uninitialized value <> for each type. The capsule evaluation rules consider a variable to be irreducible if it is bound to this value. The variable can be used in computations as long as there

is no attempt to deconstruct it; any such attempt results in a runtime error. "Deconstruction" here means different things for different types. For a coinductive type, it means applying a destructor. For `int`, it would mean attempting to perform arithmetic with it. But it can be used as the argument of a constructor or can appear on the left-hand side of an assignment without error, as these do not require deconstruction. This allows coalgebraic values and recursive functions to be created in a uniform way via backpatching (a.k.a. Landin's knot). Thus, `let rec x = d in e` is syntactic sugar for

```
let x = <> in (x := d);e
```

which in turn is syntactic sugar for

```
(fun x -> (x := d);e) <>
```

For example, `let rec x = (x,x) in snd (snd x)` becomes

```
let x = <> in (x := (x,x)); snd (snd x)
```

During the evaluation of `(x,x)`, the variable `x` is bound to `<>`, so `x` is not reduced. [1] The value of the expression is just `(x,x)`. Now the assignment `x := (x,x)` is performed, and `x` is rebound to the expression `(x,x)` in the environment. We have created an infinite coinductive object, namely an infinite complete binary tree. Evaluating `snd (snd x)` results in the value `(x,x)`.

Note that we never need to use placeholders or substitution to create cycles, as we are using the binding of `x` in the environment for this purpose. This is a major advantage over previous approaches [25, 41, 42, 45]. Once `x` is rebound to a non-`<>` value, it can be deconstructed after looking it up in the environment.

The variable `x` also gives a handle into the data structure that allows it to be manipulated dynamically. For example, here is a program that creates a cyclic object of length 3, then extends it to length 4:

```
> let rec x =  1 :: 2 :: 3 :: x;;
val x : int list = [1; 2; 3; 1; 2; 3; ...]
> let y = x in x := 0 :: y; x;;
- : int list = [0; 1; 2; 3; 0; 1; 2; 3; ...]
```

Any cycle must always contain at least one such variable. Two elements of a coalgebraic type are considered equal iff they are bisimilar (see §6.3). For this reason, coalgebraic types are not really the same as the circular data structures as studied in [41, 12, 25, 42]. The example above shows that we now allow run time definition of regular coinductive lists, whereas OCaml only allows infinite lists that are defined statically.

A downside to this approach is that the presence of the value `<>` requires a runtime check on value lookup. This is a sacrifice we have made to accommodate functional and imperative programming styles in a common framework, which is one of the main motivating factors behind capsules. For a basic introduction to capsule semantics, see [29], and for a full account of capsule semantics in the presence of coalgebraic types, see [32].

The reader might wonder what the connection between capsules and closures is. A closure represents a single value, whereas a capsule represents the entire global state of a computation, with no need to use heaps, stacks, or any notion of global store. The precise relationship between capsules and closures is explained in [28].

[1] Actually, this is not quite true—a fresh variable is substituted for x by α-conversion first. But we ignore this step to simplify the explanation.

3.3. Recursive functions on regular coinductive types

A general picture on how a recursive function $h : C \to A$ is defined is given by the commuting diagram

$$\begin{array}{ccc} C & \xrightarrow{\quad h \quad} & A \\ {\scriptstyle\gamma}\downarrow & & \uparrow{\scriptstyle\alpha} \\ FC & \xrightarrow{\quad Fh \quad} & FA \end{array} \tag{1}$$

Here, F is a functor that determines the structure of the base cases and recursive calls. The function $\gamma : C \to FC$ on input $x \in C$ tests for the base cases, and in the recursive case, prepares the arguments for the recursive calls. The function $Fh : FC \to FA$ performs the recursive calls and $\alpha : FA \to A$ takes the values from the recursive calls and assembles them into the value $h(x)$.

Ordinary recursively defined functions on well-founded datatypes (or in other words datatypes defined as the initial algebra of a given functor) fall into this framework. Indeed, if the domain C is the initial F-algebra then we immediately know by initiality that h is unique, since it is a map into another F-algebra. Another (dual) setting in which is easy to obtain h uniquely is if the codomain is a final coalgebra.

This general idea has been well studied [3, 4, 14, 18, 43]. Most of that work is focused on conditions ensuring unique solutions, primarily when the domain C is well-founded or when the codomain A is a final coalgebra. Also closely related are the work of Widemann [45] on coalgebraic semantics of recursion and cycle detection algorithms, the work on coinductive logic programming [39, 40, 7] and the work on coinductive featherweight Java [8, 9], which address many of the same issues but in the context of logic and object-oriented programming.

As we mentioned above, ordinary recursion over inductive datatypes corresponds to the case in which C is well-founded. In this case, the solution h always exists and is unique. However, if C is not well-founded, then the solution may not be unique, and the one given by the standard semantics of recursive functions is usually not the one we want. Nevertheless, the diagram (1) can still serve as a valid definitional scheme, provided we are allowed to specify an alternative solution method in A for the equations defined by the diagram above. This has been object of study in the 2006 paper by Adamek, Milius and Velebil [4] and in the recent papers by Jeannin, Kozen and Silva [30, 31].

The free variables example from §2 fits this precisely this scheme. Instantiating the diagram above yields:

$$\begin{array}{ccc} \mathtt{Term} & \xrightarrow{\quad \mathtt{fv} \quad} & \mathcal{P}(\mathtt{Var}) \\ {\scriptstyle\gamma}\downarrow & & \uparrow{\scriptstyle\alpha} \\ F(\mathtt{Term}) & \xrightarrow[\mathsf{id}_{\mathtt{Var}} + \mathtt{fv}^2 + \mathsf{id}_{\mathtt{Var}} \times \mathtt{fv}]{} & F(\mathcal{P}(\mathtt{Var})) \end{array}$$

where $FX = \texttt{Var} + X^2 + \texttt{Var} \times X$ and

$$\begin{aligned} \gamma(\texttt{Var}\ x) &= \iota_0(x) & \alpha(\iota_0(x)) &= \{x\} \\ \gamma(\texttt{App}\ (t_1, t_2)) &= \iota_1(t_1, t_2) & \alpha(\iota_1(u, v)) &= u \cup v \\ \gamma(\texttt{Lam}\ (x, t)) &= \iota_2(x, t) & \alpha(\iota_2(x, v)) &= v \setminus \{x\}. \end{aligned}$$

Here the domain (regular λ-coterms) is not well-founded and the codomain (sets of variables) is not a final coalgebra, but the codomain is a CPO under the usual set inclusion order with bottom element $\varnothing$, and the desired solution is the least solution in this order; it is just not the one that would be computed by the standard semantics of recursive functions. In such cases our language allows the programmer to specify an alternative solution method implemented by a solver, like least fixpoint computation.

4. Equations and Solvers

When the programmer makes a function call $f(a_0)$, where f was defined using the `corec` keyword, execution happens in three distinct steps:

- a set of *equations* is generated;
- the equations are sent to a solver; the solver can be built-in or user-defined;
- the result of running the solver on the set of equations is returned as the result of function call $f(a)$.

In this section we describe in detail how equations are generated, and the different possible choices for the solver.

4.1. Equation Generation

An *equation* denotes an equality between two terms. Its left-hand-side is a variable x_i that stands for the call of f on some input a_i. Its right-hand-side is a partially evaluated abstract syntax tree: it is an expression of the language which can contain other variables x_j.

When calling a recursive function f on an inductive (well-founded) term a_0, this function can make recursive calls, generating new calls to function f. The reason this computation finishes is because the computation is well-founded: every path in the call tree reaches a base case.

Similarly, if the function f was defined with the `corec` keyword, its call on a coinductive term a_0 might involve some recursive calls; those recursive calls might themselves involve some recursive calls, and so on. This time the computation is not well-founded, but because a_0 has a finite representation, the set of possible such calls is finite, for example recursive calls were made on $a_1, \ldots, a_n$.

While executing those recursive calls, a fresh variable x_i is generated for each a_i, and the call to $f(a_i)$ is partially evaluated to generate an equation, replacing the calls to $f(a_j)$ by their corresponding x_j. We thus generate a set of equations whose solution is the value of $f(a_0)$. Of course, the arguments $a_0, \ldots, a_n$ are not known in advance, so the x_i have to be generated while the program is exploring the recursive calls. This is achieved by keeping track of all the a_i that have been seen so far, along with their associated unknowns x_i.

A *solver* takes a set of equations and returns a solution, or fails. We have currently implemented four built-in solvers, three of which are quite versatile and can be used in many different applications. We also give to the programmer the ability to define his/her own solvers.

4.2. The `iterator` Solver

In many cases the set of equations can be seen as defining a fixpoint of a monotone function. For example, when the codomain is a CPO, and the operations on the right-hand sides of the equations are monotone, then the Knaster–Tarski theorem ensures that there is a least fixpoint. Moreoever, if the CPO is finite or otherwise satisfies the ascending chain condition (ACC), then the least fixpoint can be computed in finite time by iteration, starting from the bottom element of the CPO.

The `iterator` solver takes an argument b representing the initial guess for each unknown. In the case of a CPO, this would typically be the bottom element.

Internally, a guess is made for each unknown, initially b. At each iteration, a new guess is computed for each unknown by evaluating the corresponding right-hand side, where the unknowns have been replaced by current guesses. When all the new guesses equal the old guesses, we stop, as we have reached a fixpoint, the intended result. The right-hand sides are evaluated in postfix order, i.e., in the reverse order of seeing and generating new equations, because it usually makes the iteration converge faster.

Note that this `iterator` solver is closely related to the least fixpoint solver described in [30, 35], but it can also be used in applications where the desired fixpoint is not necessarily the least.

Example. We revisit the example from the introduction by applying this solver to create a function `set` that computes the set of all elements appearing in a list. A regular list, even if it is infinite, has only finitely many elements. If A is the type of the elements, the codomain of `set` is the CPO $(\mathcal{P}(A), \subseteq)$ with bottom element $\varnothing$. Restricted to subsets of the set of variables appearing in the list, it satisfies the ascending chain condition, which ensures that the least fixed point can be computed in finite time by iteration.

For the implementation, we represent a set as an ordered list. The function `insert` inserts an element into an ordered list without duplicating it if it is already there. The function `set` can be defined as:

```
let corec[iterator []] set l = match l with
    | [] -> []
    | h :: t -> insert h (set t)
```

The complexity of this solver depends on the number of iterations; at each iteration every equation is evaluated, which leads to a complexity on the order of the product of the number of iterations by the number of equations.

4.3. The `constructor` Solver

The `constructor` solver can be used when a function tries to build a data structure that could be cyclic, representing a regular coinductive element. Internally, `constructor` first checks that the right-hand side of every equation is a value (an integer, float, string, Boolean, unit, tuple on values or unknowns, injection on a value or unknown). Then it replaces the unknown variables on the right-hand sides with normal variables and adds them to the environment, thus creating the capsule representing the desired data structure. Its complexity is linear in the number of equations.

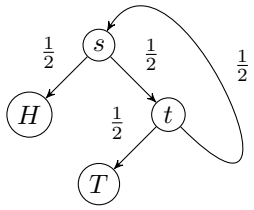
Example. The `map` function on lists takes a function `f` and a list `l`, applies `f` on every element `h` of `l`, and returns the list of the results `f h`. The `constructor` solver can be used to create a `map` functions that works on all lists, finite or infinite.

```
let corec[constructor] map arg = match arg with
   | f, [] -> []
   | f, h :: t -> f(h) :: map (f,t)
```

4.4. The `gaussian` Solver

The `gaussian` solver is designed to be used when the function computes a linear combination of recursive calls. The set of equations is then a Gaussian system that can be solved by standard techniques.

Example. Imagine one wants to simulate a biased coin, say a coin with probability $2/3$ of heads, with a fair coin. Here is a possible solution: flip the fair coin. If it comes up heads, output heads, otherwise flip again. If the second flip is tails, output tails, otherwise repeat from the start. This protocol can be represented succinctly by the following probabilistic automaton:



Operationally, starting from states s and t, the protocol generates series that converge to $2/3$ and $1/3$, respectively.

$$\mathsf{Pr}_H(s) = \tfrac{1}{2} + \tfrac{1}{8} + \tfrac{1}{32} + \tfrac{1}{128} + \cdots = \tfrac{2}{3}$$
$$\mathsf{Pr}_H(t) = \tfrac{1}{4} + \tfrac{1}{16} + \tfrac{1}{64} + \tfrac{1}{256} + \cdots = \tfrac{1}{3}.$$

However, these values can also be seen to satisfy a pair of mutually recursive equations:

$$\mathsf{Pr}_H(s) = \tfrac{1}{2} + \tfrac{1}{2} \cdot \mathsf{Pr}_H(t) \qquad\qquad \mathsf{Pr}_H(t) = \tfrac{1}{2} \cdot \mathsf{Pr}_H(s).$$

In CoCaml, we can model the automaton by a coinductive type and define a function computing the probability of Heads using the `gaussian` solver:

```
type tree = Heads | Tails
          | Flip of float * tree * tree

let corec[gaussian] probability t = match t with
     Heads -> 1.
   | Tails -> 0.
   | Flip(p, t1, t2) -> p *. probability t1 +.
                        (1. -. p) *. probability t2
```

4.5. The `separate` Solver

In both OCaml and CoCaml, the default printer for lists prints up to some preset depth, printing "..." when this depth is exceeded. This will always happen if the list is circular.

```
let rec ones = 1 :: ones;;
val ones : int list = [1; 1; 1; 1; 1; 1; 1; ...]
```

This is not very satisfying. Often it may appear as if some pattern is repeating, but what if for instance a 2 appears in 50th position and is not printed? A better solution might be to print the non-repeating part normally, followed by the repeating part in parenthesis. For example, the list `[1; 2]` might be printed `12` and the list `1 :: 2 :: ones` be printed `12(1)`. This can be achieved by creating a special solver `separate`, which from the equations defining the lists outputs two finite lists, the non-repeating part and the repeating part. From there it is easy to finish.

Internally, the equations given to the solver are a graph representing the list. A simple cycle-detection algorithm allows us to solve the equations as desired. Its complexity is linear in the number of equations.

```
type 'a sep = I1 | I2 of 'a * sep | I3 of 'a list * 'a list
let corec[separate] separate i = match i with
      [ ] -> I1
    | i :: t -> I2(i, separate t)
```

However, this example is not completely satisfying. In fact, the function `separate` seems to not be doing anything, and the solver is quite ad hoc, which contrasts greatly with the solvers we have seen so far. Moreover, the type `sep` exists merely to make the type checker happy: the function looks like it returns an element injected with `I1` or `I2`; in reality this element is passed to the solver which returns a pair of lists, injected with `I3`. Conceptually, the solver takes a list as an argument and returns a pair of lists. This example shows the limits of the current typing mechanism as applied to functions on coinductive data.

4.6. User-defined Solvers

The solvers we have presented so far are implemented directly in the interpreter. However, as versatile as these solvers are, the programmer sometimes needs to define his/her own solver. This can be done by defining a module of type `Solver`.

```
module type Solver = sig
   type var
   type expr
   type t
   val fresh : unit -> var
   val unk : var -> expr
   val solve : var -> (var * expr) list -> t
end
```

Type `var` is the type of the variables in the equations, and also the type of the left-hand sides of the equations; type `expr` is the type of the right-hand sides of the equations; and type `t` is the return type of the solver, and thus also of the function that is being defined.

Function `fresh` generates fresh elements of type `var`, it is called on each element of the coalgebra that is encountered; it is the responsibility of the user to provide a function that generates elements of

type `var` that are all different. In most cases type `var` is simply `string`, and fresh strings can easily be generated, for example with the function:

```
let fresh = let c = 0 in
   (fun (x:unit) -> c := c+1;
                    "fresh" @ (string_of_int c))
```

But the programmer could choose a different type `var`, for instance to store more information in it.

To represent variables on the right-hand sides of equations, we need to be able to inject an element of type `var` into the type `expr`. Function `unk` provides this. Typically `expr` is a sum type that contains a special case `Unknown of var`, and the injection is just `fun x -> Unknown x`.

Finally, `solve` is the solver itself. By construction, an equation always has a variable on its left-hand side, and an `expr` on its right-hand side, and is thus represented as a pair of type `var * expr`. Given an element x of type `var`, and a set (represented as a list) of equations, it returns an element of type `t` that is a solution for the variable x satisfying those equations.

Example. We could define the gaussian solver, as a user-defined solver `Gaussian` by taking `var = string`, `t = float`, and

```
type expr =
     Val of float
   | Plus of expr * expr
   | Minus of expr * expr
   | Mul of expr * expr
   | Unknown of var
```

`fresh` and `unk` are the typical ones shown above, and `solve` implements a gaussian-elimination algorithm in CoCaml. In the definition of the type `expr`, we could have chosen to have `Mul` with arguments `float*expr` which would automatically keep the equations linear. We write instead `Mul of expr*expr` which is more general, and we check linearity dynamically. The declaration of function `probability` becomes

```
let corec[Gaussian] probability t = match t with
     Heads -> Val 1.
   | Tails -> Val 0.
   | Flip(p, t1, t2) -> Plus(Mul(Val p, probability t1),
                             Mul(Val (1.-. p), probability t2))
```

The right-hand side is slightly different and less clear than in the original definition. In some sense, instead of working with the abstract syntax tree of the whole language, the programmer is able to define his/her own small abstract syntax tree to work with, representing right-hand sides of functions. This is reminiscent of a known technique to solve corecursive equations by defining a coalgebra whose carrier is a set of expressions comprising the intermediate steps of the unfolding of the equations [37, 38, 13].

5. Examples

In this section, we show several examples of functions on coinductive types, including finite and infinite lists, a library for p-adic numbers, and infinitary λ-terms.

5.1. Finite and Infinite Lists

We present an application of our main solvers through examples on lists, one of the simplest examples of coinductive types. Through these examples, we show how easy it is to create recursive functions on regular coinductive datatypes, as the process is very close to creating recursive functions on inductive datatypes.

5.1.1. Test of Finiteness

We would like to be able to test whether a list is finite or infinite. The most intuitive way of doing this is to write a function like:

```
let rec is_finite l = match l with
   | [] -> true
   | h :: t -> is_finite t
```

Of course, this does not terminate on infinite lists under the standard semantics of recursive functions. However, if we use the `corec` keyword, the equations generated for `[0]` will look like

```
is_finite [0] = is_finite []
is_finite [] = true
```

and the result will be `true`. For the infinite list `ones`, the only equation will look like

```
is_finite(ones) = is_finite(ones)
```

and we expect the result to be `false`. Intuitively, the result of solving the equations should be `true` if and only if the expression `true` appears on the right-hand side of one of the equations. This can be achieved with the iterator solver, using as first guess the value we should observe if it is not finite, here `false`:

```
let corec[iterator false] is_finite = function
   | [] -> true
   | h :: t -> is_finite t
```

5.1.2. List `exists`

Given a Boolean-valued function `f` that tests a property of elements of a list `l`, we would like to define a function that tests whether this property is satisfied by at least one element of `l`. The function can simply be programmed using the `iterator` solver, where the default value should be `false`:

```
let corec[iterator false] exists arg = match arg with
   | f, [] -> false
   | f, h :: t -> f(h) || exists (f, t)
```

Note that for this function to work, it is critical that the "or" operator `||` be lazy, so that the partial evaluation of the expression `f(h) || exists (f, t)` can return `true` directly whenever `f(h)`, even if the result of evaluating `exists(f, t)` is not known.

5.1.3. The Curious Case of Filtering

Given a Boolean-valued function function `f` and a list `l`, we would like to define a function that creates a new list `l1` by keeping only the elements of `l` that satisfy `f`. The first approach is to use the `constructor` solver and do it as if the list were always finite:

```
let corec[constructor] filter_naive arg = match arg with
   | f, [] -> []
   | f, h :: t -> if f(h) then h :: filter_naive(f, t)
                  else filter_naive(f, t)
```

However, this does not quite work. For example, if called on the function `fun x -> x <= 0` and the list `ones`, it generates only one equation

```
filter_naive(ones) = filter_naive(ones)
```

and it is not clear which solution is desired by the programmer. However, it is clear that in this particular case, the list `[]` should be returned. The problem arises whenever the function is called on an infinite list `l` such that no element of `l` satisfies `f`. Rather than modify the solver, our solution is to be a little bit more careful and return `[]` explicitly when needed:

```
let corec[constructor] filter arg = match arg with
   | f, [] -> []
   | f, h :: t: -> if f(h) then h :: filter(f, t)
                   else if exists(f, t) then filter(f, t)
                   else []
```

The main problem with this solution is that it has a quadratic complexity in the size of the internal representation of its second argument — if considering each call to its first argument `f` to be $O(1)$. It is possible to get a linear complexity using an ad hoc solver, not presented here.

5.1.4. Other Examples on Lists

We have presented a few examples of functions on infinite lists. Some of them are inspired by classic functions on lists supported by the List module of OCaml. Some functions of the List module, like sorting, do not make sense on infinite lists. But most other functions of the List module can be implemented in similar ways. We refer to the implementation provided as attachment for more details.

5.2. A Library for p-adic Numbers

In this section we present a library for p-adic numbers and operations on them.

5.2.1. The p-adic Numbers

The p-adic numbers [10, 46] are a well-studied mathematical structure with applications in several areas of mathematics. For a fixed prime p, the p-adic numbers $\mathbb{Q}_p$ form a field that is the completion of the rationals under the p-adic metric in the same sense that the reals are the completion of the rationals under the usual Euclidean metric. The p-adic metric is defined as follows. Define $|\cdot|_p$ by

- $|0|_p = 0$;

- if $x \in \mathbb{Q}$, write x as $x = ap^n/b$, where n, a and b are integers and neither a nor b is divisible by p. Then $|x|_p = p^{-n}$.

The distance between x and y in the p-adic metric is $|x - y|_p$. Intuitively, x and y are close if their difference is divisible by a high power of p.

Just as a real number has a decimal representation with a finite number of nonzero digits to the left of the decimal point and a potentially infinite number of nonzero digits to the right, a p-adic number has a representation in base p with a finite number of p-ary digits to the right and a potentially infinite number of digits to the left. Formally, every element of $\mathbb{Q}_p$ can be written in the form $\sum_{i=k}^{\infty} d_i p^i$, where the d_i are integers such that $0 \leq d_i < p$ and k is an integer, possibly negative. An important fact is that this representation is unique (up to leading zeros), in contrast to the decimal representation, in which $1 = 0.999\ldots$. If $d_k = 0$ for $k < 0$, then the number is said to be a *p-adic integer*. If b is not divisible by p, then the rational number a/b is a p-adic integer. Finally, p-adic numbers for which the sequence $(d_k)_k$ is regular (ultimately periodic) are exactly the rational numbers. This is similar to the decimal representations of real numbers. Since our lists must be regular so that they can be represented in finite memory, these are the numbers we are interested in. We fix the prime p (written `p` in programs) once and for all, for instance as a global variable.

5.2.2. Equality and Normalization

We represent a p-adic number $x = \sum_{i=k}^{\infty} d_i p^i$ as a pair of lists:

- the list $d_0, d_1, d_2, \ldots$ in that order, which we call the *integer part* of x and which can be finite or infinite; and
- if $k < 0$ and $d_k \neq 0$, the list containing $d_{-1}, d_{-2}, \ldots, d_k$, which we call the *fractional part* of x and which is always finite.

Since the representation $x = \sum_{i=k}^{\infty} d_i p^i$ is unique up to leading zeros, the only thing we have to worry about when comparing two p-adic integers is that an empty list is the same as a list of zeros, finite or infinite. The following function `equali` uses the `iterator` solver and compares two integer parts of p-adic numbers for equality:

```
let corec[iterator true] equali p = match p with
     [], [] -> true
   | h1 :: t1, h2 :: t2 -> h1 = h2 && equali (t1, t2)
   | 0 :: t1, [] -> equali (t1, [])
   | [], 0 :: t2 -> equali (t2, [])
   | x -> false
```

Interestingly, comparing the fractional parts is almost the same code, with the **`rec`** keyword instead of the `corec` keyword.

```
let rec equalf p = match p with (* of floating parts: not corecursive *)
     [], [] -> true
   | h1 :: t1, h2 :: t2 -> h1 = h2 && equalf (t1, t2)
   | 0 :: t1, [] -> equalf (t1, [])
   | [], 0 :: t2 -> equalf (t2, [])
   | _ -> false
```

```
let equal p1 p2 = match p1, p2 with
   (i1, j1), (i2, j2) -> equali (i1, i2) && equalf (j1, j2)
```

This happens quite often: if one knows how to do something with inductive types, the solution for coinductive types often involves only changing the **rec** keyword to `corec` and some other minor adjustments. However, one must take care, as there are exceptions to this rule. In this example, since here `equali` also works on inductive types, we could have used `equali` instead of `equalf` in `equal`.

Now that we have equality, normalization of a p-adic integer becomes easy using the `constructor` solver:

```
let corec[constructor] normalizei i =
   if equali(i, []) then []
   else match i with i :: t -> i :: normalizei t
```

The function `normalizei` only requires equality with zero (represented as `[]`), which is much easier than general equality. We can now write a normalization on the fractional parts as a simple recursive function (once again, with the same code), or just use `normalizei`, which also works on the fractional parts.

5.2.3. Conversion from a Rational

We wish to convert a given rational a/b with $a, b \in \mathbb{Z}$ to its p-adic representation. Let us first try to convert $x = a/b$ into a p-adic integer if b is not divisible by p. Since x is a p-adic integer, we know that x can be written $x = \sum_{i=0}^{\infty} d_i p^i$, thus multiplying both sides by b gives

$$a = b \sum_{i=0}^{\infty} d_i p^i.$$

Taking both sides modulo p, we get $a = bd_0 \bmod p$. Since b and p are relatively prime, this uniquely determines d_0 such that $0 \leq d_0 < p$, which can be found by the Euclidean algorithm. We can now substract bd_0 to get

$$a - bd_0 = b \sum_{i=1}^{\infty} d_i p^i.$$

This can be divided by p by definition of d_0, which leads to the same kind of problem recursively.

This procedure defines an algorithm to find the digits of a p-adic integer. Since we know it will be cyclic, we can use the `constructor` solver:

```
let corec[constructor] from_rationali (a,b) =
   if a = 0 then []
   else let d = euclid p a b in
      d :: from_rationali ((a - b*d)/p, b)
```

where the call `euclid p a b` is a recursive implentation of a (slightly modified) Euclidean algorithm for finding d_0 as above.

If b is divisible by p, it can be written $p^n b_0$ where b_0 is not divisible by p, and we can first find the representation of a/b_0 as an integer, then shift by n digits to simulate division by p^n.

5.2.4. Conversion to a Float

Given a p-adic integer $x = \sum_{i=0}^{\infty} d_i p^i$, define $x_k = \sum_{i=0}^{\infty} d_{k+i} p^i$. Then for all $k \geq 0$, $x_k = d_k + p x_{k+1}$. If the sequence $(d_k)_k$ is regular, so is the sequence $(x_k)_k$, thus there exist $n, m > 0$ such that $x_{k+m} = x_k$ for all $k \geq n$. It follows that

$$x = x_0 = \sum_{i=0}^{n-1} d_i p^i + p^n x_n \qquad\qquad x_n = \sum_{i=0}^{m-1} d_{n+i} p^i + p^m x_n,$$

and further calculation reveals that $x = a/b$, where

$$a = \sum_{i=0}^{n+m-1} d_i p^i - \sum_{i=0}^{n-1} d_i p^{m+i} \qquad\qquad b = 1 - p^m.$$

But even without knowing m and n, the programmer can write a function that will automatically construct a system of $m + n$ linear equations $x_k = d_k + p x_{k+1}$ in the unknowns $x_0, \ldots, x_{m+n-1}$ and solve them by Gaussian elimination to obtain the desired rational representation. To accomplish this, we can just use our `gaussian` solver:

```
let corec[gaussian] to_floati i = match i with
   | [] -> 0.
   | d :: t -> (float_of_int d) +.
               (float_of_int p) *. (to_floati t)
```

This function returns the floating point representation of a given p-adic integer. It is interesting to note that, apart from the mention of `corec[gaussian]`, this is exactly the function we would have written to calculate the floating-point value of an integer written in p-ary notation using Horner's rule.

A similar program can be used to convert the floating part of a p-adic number to a float. Adding the two parts gives the desired result.

5.2.5. Addition

Adding two p-adic integers is suprisingly easy. We can use (a slight adaptation of) the primary school algorithm of adding digit by digit and using carries. A carry might come from adding the floating parts, so the algorithm really takes three arguments, the two p-adic integers to add and a carry. Using the `constructor` solver, this gives:

```
let corec[constructor] addi arg = match arg with
   | [], [], c ->
       if c = 0 then []
       else (c mod p) :: addi ([], [], c/p)
   | h :: t, [], c ->
        addi (h :: t, [0], c)
   | [], h :: t, c ->
        addi ([0], h :: t, c)
   | hi :: ti, hj :: tj, c ->
       let res = hi + hj + c in
       (res mod p) :: addi (ti, tj, res / p)
```

Once again, once we have addition on p-adic integers, it is easy to program addition on general p-adic numbers.

5.2.6. Multiplication and Division

The primary school algorithm and the `constructor` solver can also be used for multiplication. However, we need to proceed in two steps. We first create a function `mult1` that takes a p-adic integer `i`, a digit `j`, and a carry `c`, and calculates `i*j+c`. We then create a function `multi` that takes two p-adic integers `i` and `j` and a carry `c` and calculates `i*j+c`.

```
let corec[constructor] mult1 arg = match arg with
   | [], d, c -> if c = 0 then []
                 else (c mod p) :: mult1 ([], d, c/p)
   | hi :: ti, d, c ->
          let res = hi * d + c in
          (res mod p) :: mult1 (ti, d, res / p)

let corec[constructor] multi arg = match arg with
   | n1, [], c -> c
   | n1, h2 :: t2, c ->
      (match (addi (mult1 (n1, h2, 0), c, 0)) with
          | [] -> 0 :: multi (n1, t2, 0)
          | hr :: tr -> hr :: multi (n1, t2, tr) )
```

To extend this to general p-adic numbers, we can multiply both `i` and `j` by suitable powers of p before applying `multi`, then divide the result back as necessary.

Division of p-adic integers can be done with only one function using a `constructor` solver in much the same way as addition or multiplication. The algorithm also uses the `euclid` function and is closely related to `from_rational`.

Some of the examples above on p-adic numbers could be done with lazy evaluation, namely the arithmetic operations and conversion to a rational. However equality, normalization, conversion to a float and printing (showing the cycle) could not.

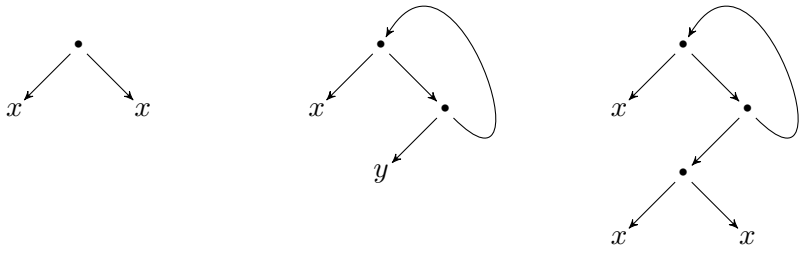
5.3. Equality

Now that we have recursive functions on coinductive types, we might ask whether it would be possible to program equality on a coinductive type. The answer is yes. The function is built in much the same way as the `equali` function, with the `iterator true` solver. This is a general trend for coinductive equality: two elements are equal unless there is evidence that they are unequal.

The code can be found below, with the small simplification of having expressions on pairs instead of general tuples. The argument to the `equal` function is a capsule, whose environment is represented as an association list. The function `assoc` is similar to the standard OCaml function `List.assoc` and looks up an element in an association list, returning its associated value.

```
type expr =
   | Var of string
   | Int of i
   | Inj of string * expr
   | Pair of expr * expr

let corec[iterator true] equal arg = match arg with
     (Var x1, env1), (Var x2, env2) ->
```



(a) (b) (c)

Figure 1. A substitution example.

```
      equal ((assoc (x1, env1), env1), (assoc (x2, env2), env2))
| (Var x1, env1), s2 -> equal ((assoc (x1, env1), env1), s2)
| s1, (Var x2, env2) -> equal (s1, (assoc (x2, env2), env2))
| (Int i1, env1), (Int i2, env2) -> i1 = i2
| (Inj(inj1, e3), env1), (Inj(inj2, e4), env2) ->
      inj1 = inj2 && equal ((e3, env1), (e4, env2))
| (Pair(e1, e2), env1), (Pair(e3, e4), env2) ->
      equal((e1, env1), (e3, env1)) && equal((e1, env1), (e3, env1))
| _ -> failwith "type error"
```

5.4. Other Examples

Besides the examples presented here, we have implemented all the examples that Jeannin, Kozen and Silva collected in [30] to CoCaml. Among those, substitution in a infinite λ-term (λ-coterm) and descending sequences can be implemented with the `constructor` solver; free variables of a λ-coterm and abstract interpretation of while loops can be implemented using the `iterator` solver; and the examples involving probabilistic protocols, like calculating the probability of heads of a coin-flip protocol or the expected number of flips, can be implemented using the `gaussian` solver. Further examples include algorithms for finite automata and abstract interpretation.

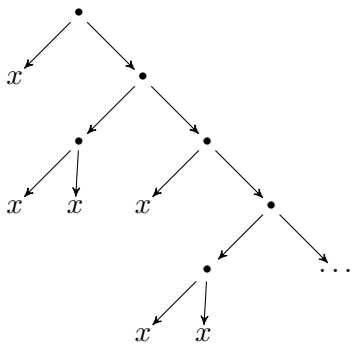
To complement the example of in §2 involving the free variables of a λ-coterm, we show another non-well-founded example on λ-coterms, namely the substitution of a term `t` for all free occurrences of a variable `y`. A typical implementation would be

```
let rec subst t y = function
   | Var x -> if x = y then t else Var x
   | App (t1,t2) -> App (subst t y t1, subst t y t2)
```

For simplicity, we have omitted the case of function abstraction, since it is not relevant for the example.

For example, to replace y in Fig. 1(b) by the term of Fig. 1(a) to obtain Fig. 1(c), we would call `subst (App x x) y t`, where `t` is the term of Fig. 1(b), defined by `let rec t = App x (App y t)`.

The usual semantics would infinitely unfold the term on the left, attempting to generate



This computation would never finish.

A minor adaptation of the definition of `subst` above results in a function that will return the desired regular infinitary λ-term:

```
let corec[constructor] subst arg = match arg with
   | x, t, Var v -> if v = x then t else Var v
   | x, t, App(t1, t2) -> App(subst (x, t, t1), subst (x, t, t2))
```

6. Implementation

We have implemented an interpreter for CoCaml. The implementation is about 5000 lines of OCaml. We have also used CoCaml on a number of examples, of which we have presented a few here. So far we have written about 1500 lines of CoCaml. The implementation is provided in the supplementary material.

6.1. Overview

In this section, we explain how the translation of the construct `corec[solver]` is implemented. Briefly, the body of the function is partially evaluated, replacing recursive calls by variables, and this forms the right-hand side of an equation. The generated equations are then solved using the parameter `solver`.

Note that equations can only be correctly generated if all the recursive calls are applied to an argument, and none of them are nested. This argument needs to be explicit, and examples such as

```
let corec[constructor] alternating_bools =
   false :: map not alternating_bools
```

are thus ruled out. Right-hand sides can contain calls to previously defined `corec` functions as long as they are not nested.

When a function f is defined using the `corec` keyword, it is bound in the current environment. For simplicity, we impose the restriction that f takes only one argument (by forbidding curried definitions with the `corec` keyword). This is a mild restriction, as this argument can be a tuple. Also, because of how functions defined with `corec` are evaluated, we disallow nested recursive calls to f.

The interesting part occurs when the function f is called on an argument a. Since our language is call-by-value, a is first evaluated and bound in the current environment. We then proceed to generate the recursive equations that the value of $f(a)$ must satisfy.

6.2. Partial Evaluation

Partial evaluation is much like normal evaluation except when encountering a recursive call to f. When such a recursive call $f(a_j)$ is encountered, its argument is evaluated, and the call is replaced by a variable x_j corresponding to a_j. The variable x_j might be fresh if a_j had not been seen before, or it might be the one already associated with a_j.

Coming back up the abstract syntax tree, some operations cannot be performed. If the condition of an **`if`** statement was only partially evaluated, we cannot know which branch to evaluate next; the same thing happens for the condition of a **`while`** loop or an argument that is pattern-matched.

Particular care must be taken when evaluating the `&&` and `||` constructs. These are usually implemented lazily. If the first argument of `&&` evaluates to `false`, then it should return `false`. But if it only partially evaluates, then the second argument cannot be evaluated. However, we choose to partially evaluate it anyway, in case it contains recursive calls; thus our implementations of `&&` and `||` in the partial evaluator are not strictly lazy.

6.3. Equality of Regular Coinductive Terms

Equality of values, and in particular equality of cyclic data structures, plays a central role in the process of generating the equations corresponding to the call of a recursive function. A new equation is generated for each recursive call whose argument has not been previously seen. To assess whether the argument has been previously seen, a set of objects previously encountered is maintained. At each new recursive call, the argument is tested for membership in this set by testing equality with each member of the set. To ensure termination, equality on values that are observationally equivalent must return true.

Unfortunately, OCaml's documentation tells us that "equality between cyclic data structures may not terminate." In practice, the OCaml equality test returns `false` if it can find a difference in finite time, otherwise continues looping forever. In short, it never returns `true` when the arguments are cyclic and bisimilar.

```
> let rec zeros = 0 :: zeros and ones = 1 :: ones;;
val zeros : int list = [0; 0; 0; 0; 0; 0; 0; ...]
val ones : int list = [1; 1; 1; 1; 1; 1; 1; ...]
> zeros = ones;;
- : bool = false
> zeros = zeros;; (* does not terminate *)
> let rec zeros2 = 0 :: 0 :: zeros2;;
val zeros2 : int list = [0; 0; 0; 0; 0; 0; 0; ...]
> zeros = zeros2;; (* does not terminate *)
```

We would like to create a new equality, simply denoted =, that would work the same as in OCaml on every value except cyclic data structures. On cyclic data structures, this equality should correspond to observational equality, so that both calls `zeros = zeros` and `zeros = zeros2` above should return `true`. Note that the OCaml physical identity relation == is not suitable: `zeros == zeros2` would return `false`. More importantly, even two instances of a pair of integers formed at different places in the program would not be equal under ==, although they are observationally equivalent.

To allow cyclic data structures and recursive functions, values are represented internally with capsules. We are thus interested in creating observational equality on capsules. Recall that capsules are

essentially finite coalgebras, finite coalgebraic representations of a regular closed λ-coterm. Let us describe the equality algorithm on a simplification of our language where value expressions can only be variables, literal integers, injections into a sum type or tuples.

Let Cap be the set of capsules. The domain of the equality is the set of pairs of capsules – $\mathsf{Cap}^2 = \mathsf{Cap} \times \mathsf{Cap}$. The codomain is the two-element Boolean algebra $\mathbb{2}$. The diagram (1) is instantiated to

$$\begin{array}{ccc} \mathsf{Cap}^2 & \xrightarrow{\quad h \quad} & \mathbb{2} \\ \gamma \downarrow & & \uparrow \alpha \\ \mathbb{2} + \mathsf{Cap}^2 + \mathsf{list}\,(\mathsf{Cap}^2) & \xrightarrow[\mathsf{id}_{\mathbb{2}} + h + \mathsf{map}\ h]{} & \mathbb{2} + \mathbb{2} + \mathsf{list}\ \mathbb{2} \end{array}$$

where the functor is $FX = \mathbb{2} + X + \mathsf{list}\ X$. Here, $\mathsf{list}\ X$ denotes lists of elements of type X, and the map function iterates a function over a list, returning a list of the results.

The function γ matches on the first component of each capsule distinguishing between the base cases and then ones in which equality needs to be recursively determined. If they are both literal integers, it returns $\iota_1(\texttt{true})$ if they are equal and $\iota_1(\texttt{false})$ otherwise. If either one is a variable, it looks up its value in the corresponding environment. If they are injections of e_1 and e_2, it returns $\iota_2(e_1, e_2)$. If they are tuples, it creates a list l of pairs whose nth element is the pair of the nth elements of the first and second tuple and returns $\iota_3(l)$. The function α, which processes the results of recursive calls, is the identity on the first two projections, and on $\iota_3(l)$ returns the conjunction of all Boolean values in the list l, that is $\texttt{true}$ if all the elements of l are true, $\texttt{false}$ otherwise.

The naive algorithm given by this diagram is quadratic and it compares all pairs of variables in the capsules. However it turns out that we can see the capsules as finite automata, and they are equal if and only if their corresponding automata are equivalent. There is a known $n\alpha(n)$ algorithm by Hopcroft and Karp [26], where α is the inverse of the Ackermann function.[2]

7. Related Work

Syme [42] describes the "value recursion problem" and proposes an approach involving laziness and substitution, eschewing mutability. He also gives a formal calculus for reasoning about the system, along with several examples. One major concern is with side effects, but this is not a particular concern for us. His approach is not essentially coalgebraic, as bisimilar objects are not considered equal. Whereas he must perform substitution on the circular object, we can use variable binding in the environment, as this is invisible with respect to bisimulation, which is correspondingly much simpler. He also claims that "compelling examples of the importance of value recursion have been missing from the literature". We have tried to fill the gap in this paper. Many more examples appear in other works, notably in [30] and, for instance, in work of Ancona which we discuss below.

Sperber and Thiemann [41] propose replacing ref cells with a safe pointer mechanism for dealing with mutable objects. Again, this is not really coalgebraic. They state that "ref cells, when compared

[2] Hopcroft and Karp initially thought the algorithm was linear, but the complexity was later corrected to $n\alpha(n)$

to mechanisms for handling mutable data in other programming languages, impose awkward restrictions on programming style," a sentiment with which we wholeheartedly agree.

Capsule semantics address the same issues as Felleisen's and Hieb's theories of syntactic state [20], but capsules are considerably simpler. A major advantage is the elimination of the explicit context present in [20].

Hirschowitz, Leroy, and Wells [25] suggest a safe initialization method for cyclic data structures. Again, their approach is not coalgebraic and uses substitution, which precludes further modification of the data objects once they are created.

Close to our work is the recent paper by Widemann [45], which is explicitly coalgebraic. He uses final coalgebras to interpret datatype definitions in a heap-based model with call-by-value semantics. Circular data objects are represented by cycles of pointers created by substitution. The main focus is low-level implementation of evaluation strategies, including cycle detection, and examples are mainly search problems. He also proposes a "front-end language" constructs as an important problem for future work, which is one of the issues we have addressed here.

Recently, copatterns [1, 2] have been developed to analyze and manipulate infinite data structures and coinductive datatypes, a goal similar to the one of CoCaml. However the approach taken is different, as it does not restrict to regular coinductive types. Thus some of our examples, such as free variables or probability, cannot be handled. Copatterns have an extensively developed type theory and are implemented in Agda.

The question of equality of circular data structures in OCaml has been subject of investigation in, e.g, [24], where the `cyclist` can be found. The `cyclist` library provides some functions on infinite lists in OCaml. However, this is limited to lists and does not handle any other coinductive type. Another relevant paper where functions on list in an ML like language are discussed is [15]. There is also work in Scheme [6] which defined observational equality for trees and graphs. The language constructs we apply in this paper could also be easily transferred to Scheme.

In the area of logic programming, similar challenges have been tackled. Coinductive logic programming (coLP) [39, 40] has been recently introduced as a step forward towards developing logic programs containing both finite and regular coinductive terms. The operational semantics is obtained by computing the greatest fixed point of a logic program. Inspired by coLP, Ancona and Zucca defined corecursive FeatherWeight Java (coFJ) [8] which extends FeatherWeight Java with language support for cyclic structures. In [9] they provide a translation from coFJ into coLP, clarifying the connections between the two frameworks, and, more notably, providing an effective implementation of coFJ. In [48], they define a type system for coFJ that allows the user to specify that certain methods are not allowed to return undetermined when the solution of the equation system is not unique. Ancona has also improved the state of the art on regular corecursion in Prolog [7], by extending the interpreter with a new clause that enables simpler definitions of coinductive predicates.

In the context of type theory, theorem provers like Coq and Agda allow coinductive programs and proofs. In Coq for example, one can define coinductive datatypes similar to the ones found in OCaml, and define functions and proofs on them [16, 22]. The coinductive functions that can be defined in Coq or Agda correspond to a subset of the functions that can be defined using the `constructor` solver of CoCaml. None of the functions using the other solvers presented in this paper are expressible. However, unlike CoCaml, Coq and Agda ensure that all functions terminate and all proofs are sound. For coinductive functions, this is traditionally done using a guardedness condition, a syntactic check on the body of the function or the proof [16]. This is sometimes restrictive, and less restrictive conditions have been

developed, focusing on programs in Coq [11, 22] and Agda [17], or on proofs in Coq [27, 34].

Work on cyclic structures in lazy languages can be found in e.g. [19, 21, 36, 44]. In these works, explicit modeling of back pointers (and visited nodes) is used, requiring for instance the use of nested datatypes, and no new program construct is proposed. In our work, we do not touch the datatype definition: the cyclic structure of an object is detected automatically.

The closest to our work, in the context of OCaml, is [30], where Jeannin, Kozen and Silva discuss at length what is needed to extend OCaml with language support to define functions on regular coinductive types. They provide a mock-up implementation also in a OCaml-like language. However, their proposal to specify alternative semantics (solution methods) is not generic and they do not provide any support for user defined solvers. In this paper, we give a full implementation and improve on their results by showing how alternative semantics, provided by equation solvers, can be given in a lightweight, elegant fashion. We implement several generic solvers, which are versatile enough to cover a wide range of examples, and give the user means to be able to define his/her own solver. All the examples discussed in [30] and also in [8] fit in our framework.

8. Conclusions and future work

Coalgebraic (coinductive) datatypes and algebraic (inductive) datatypes are similar in many ways. They are defined in the same way by recursive type equations, algebraic types as least (or initial) solutions and coalgebraic types as greatest (or final) solutions. Because of this similarity, one would like to program with them in the same way, by defining functions by structural recursion using pattern matching. However, because of the non-well-foundedness of coalgebraic data, it must be possible for the programmer to circumvent the standard semantics of recursion and specify alternative solution methods for recursive equations. Up to now, there has been little programming language support for this.

In this paper we have presented CoCaml, an extension of OCaml with new programming language constructs to address this issue for regular coinductive types. We have shown though numerous examples that regular coalgebraic types can be useful in many applications and that computing with them is in most cases no more difficult than computing with algebraic types. Although these alternative solution methods are nonstandard, they are quite natural and can be specified in succinct ways that fit well with the familiar style of recursive functional programming. We have a full implementation of our framework, including several generic solvers and support for user-defined solvers.

As future work, we would like to provide, in addition to the power of the user being able to define his/her own solvers, static checks that can be performed on a solver and its associated functions to ensure that the computation stays safe. Right now most of the checks are dynamic.

Equations can only be correctly generated if all the recursive calls are applied to an argument, and none of them are nested. We currently check this dynamically. However, we believe we can enforce this through an extended type system on which we are currently working.

We are currently extending the implementation to include regular coinductive objects in more structured categories like vector spaces. This would allow us to cover functions on rational streams and trees [38], broadening further the spectrum of applications and examples.

Finally, we would like to develop methods for proving the correctness of the implementation of recursive functions on coalgebraic data. This would open the door of connecting our work to very recent work on coinduction in program verification [33].

Acknowledgments

Thanks to Bob Constable, Jean-Christophe Filliâtre, Nate Foster, Edgar Friendly, Alain Girault, Fritz Henglein, Helle Hvid Hansen, Bart Jacobs, Jonathan Kimmitt, Andrew Myers, Stefan Milius, François Pottier, Ross Tate, and Baltasar Trancón y Widemann for stimulating discussions. Thanks to Hersh Mehta for help with the implementation.

References

[1] Abel, A., Pientka, B.: Wellfounded recursion with copatterns: a unified approach to termination and productivity, *ICFP*, 2013.

[2] Abel, A., Pientka, B., Thibodeau, D., Setzer, A.: Copatterns: programming infinite structures by observations, *POPL*, 2013.

[3] Adámek, J., Lücke, D., Milius, S.: Recursive Coalgebras of Finitary Functors, *Theoretical Informatics and Applications*, **41**, 2007, 447–462.

[4] Adámek, J., Milius, S., Velebil, J.: Elgot Algebras, *Log. Methods Comput. Sci.*, **2**(5:4), 2006, 1–31.

[5] Adámek, J., Milius, S., Velebil, J.: Iterative algebras at work, *Mathematical. Structures in Comp. Sci.*, **16**(6), December 2006, 1085–1131.

[6] Adams, M. D., Dybvig, R. K.: Efficient nondestructive equality checking for trees and graphs, *Proc. 13 ACM SIGPLAN Int. Conf. Functional Programming*, 2008.

[7] Ancona, D.: Regular corecursion in Prolog, *SAC* (S. Ossowski, P. Lecca, Eds.), ACM, 2012, ISBN 978-1-4503-0857-1.

[8] Ancona, D., Zucca, E.: Corecursive Featherweight Java, *FTfJP'012 - Formal Techniques for Java-like Programs*, 2012.

[9] Ancona, D., Zucca, E.: Translating Corecursive Featherweight Java in Coinductive Logic Programming, *Co-LP 2012 - A workshop on Coinductive Logic Programming*, 2012.

[10] Baker, A.: An Introduction to p-adic Numbers and p-adic Analysis, `http://www.maths.gla.ac.uk/~ajb/dvi-ps/padicnotes.pdf`, March 2011, School of Mathematics and Statistics, University of Glasgow.

[11] Bertot, Y., Komendantskaya, E.: Inductive and Coinductive Components of Corecursive Functions in Coq, *Electr. Notes Theor. Comput. Sci.*, **203**(5), 2008, 25–47.

[12] Boudol, G., Zimmer, P.: Recursion in the call-by-value lambda-calculus, *FICS* (Z. Ésik, A. Ingólfsdóttir, Eds.), NS-02-2, University of Aarhus, 2002.

[13] Capretta, V.: Coalgebras in functional programming and type theory, *Theor. Comput. Sci.*, **412**(38), 2011, 5006–5024.

[14] Capretta, V., Uustalu, T., Vene, V.: Corecursive Algebras: A Study of General Structured Corecursion, *Formal Methods: Foundations and Applications, 12th Brazilian Symp. Formal Methods (SBMF 2009)* (M. V. M. Oliveira, J. Woodcock, Eds.), 5902, Springer, Berlin, 2009.

[15] Caspi, P., Pouzet, M.: A Co-iterative Characterization of Synchronous Stream Functions, *Electr. Notes Theor. Comput. Sci.*, **11**, 1998, 1–21.

[16] Coquand, T.: Infinite Objects in Type Theory, *TYPES*, 1993.

[17] Danielsson, N. A.: Beating the Productivity Checker Using Embedded Languages, *PAR*, 2010.

[18] Eppendahl, A.: Coalgebra-to-Algebra Morphisms, *Electronic Notes in Theoretical Computer Science*, **29**, 1999.

[19] Fegaras, L., Sheard, T.: Revisiting catamorphisms over datatypes with embedded functions (or, programs from outer space), *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '96, ACM, New York, NY, USA, 1996.

[20] Felleisen, M., Hieb, R.: The Revised Report on the Syntactic Theories of Sequential Control and State, *Theoretical Computer Science*, **103**, 1992, 235–271.

[21] Ghani, N., Hamana, M., Uustalu, T., Vene, V.: Representing cyclic structures as nested datatypes, *Proc. of 7th Symp. on Trends in Functional Programming, TFP 2006* (H. Nilsson, Ed.), Univ. of Nottingham, 2006.

[22] Giménez, E.: Structural Recursive Definitions in Type Theory, *ICALP*, 1998.

[23] Goguen, J. A., Thatcher, J. W.: Initial Algebra Semantics, *15th Symp. Switching and Automata Theory*, IEEE, 1974.

[24] Grebeniuk, D.: Library `ocaml-cyclist`, `https://forge.ocamlcore.org/projects/ocaml-cyclist/`, June 2010.

[25] Hirschowitz, T., Leroy, X., Wells, J. B.: Compilation of extended recursion in call-by-value functional languages, *PPDP 2003*, 2003.

[26] Hopcroft, J. E., Karp, R. M.: *A linear algorithm for testing equivalence of finite automata*, Technical report, Cornell University, 1971.

[27] Hur, C.-K., Neis, G., Dreyer, D., Vafeiadis, V.: The power of parameterization in coinductive proof, *POPL*, 2013.

[28] Jeannin, J.-B.: Capsules and Closures, *Proc. 27th Conf. Math. Found. Programming Semantics (MFPS XXVII)* (M. Mislove, J. Ouaknine, Eds.), Elsevier Electronic Notes in Theoretical Computer Science, Pittsburgh, PA, May 2011.

[29] Jeannin, J.-B., Kozen, D.: Computing with Capsules, *J. Automata, Languages and Combinatorics*, **17**(2–4), 2012, 185–204.

[30] Jeannin, J.-B., Kozen, D., Silva, A.: Language Constructs for Non-Well-Founded Computation, *22nd European Symposium on Programming (ESOP 2013)* (M. Felleisen, P. Gardner, Eds.), 7792, Springer, Rome, Italy, March 2013.

[31] Jeannin, J.-B., Kozen, D., Silva, A.: *Well-Founded Coalgebras, Revisited*, Technical Report `http://hdl.handle.net/1813/33330`, Computing and Information Science, Cornell University, May 2013.

[32] Kozen, D.: New, *Proc. 28th Conf. Math. Found. Programming Semantics (MFPS XXVIII)* (U. Berger, M. Mislove, Eds.), Elsevier Electronic Notes in Theoretical Computer Science, Bath, England, June 2012.

[33] Leino, K. R. M., Moskal, M.: *Co-induction Simply: Automatic Co-inductive Proofs in a Program Verifier*, Technical report, Microsoft Research, 2013.

[34] Niqui, M.: Coalgebraic Reasoning in Coq: Bisimulation and the lambda-Coiteration Scheme, *TYPES*, 2008.

[35] Pottier, F.: Lazy Least Fixed Points in ML, Available from `pauillac.inria.fr/~fpottier/publis/fpottier-fix.pdf`.

[36] d. S. Oliveira, B. C., Cook, W. R.: Functional programming with structured graphs, *ICFP* (P. Thiemann, R. B. Findler, Eds.), ACM, 2012, ISBN 978-1-4503-1054-3.

[37] Silva, A., Rutten, J. J. M. M.: Behavioural Differential Equations and Coinduction for Binary Trees, *WoLLIC* (D. Leivant, R. J. G. B. de Queiroz, Eds.), 4576, Springer, 2007, ISBN 978-3-540-73443-7.

[38] Silva, A., Rutten, J. J. M. M.: A coinductive calculus of binary trees, *Inf. Comput.*, **208**(5), 2010, 578–593.

[39] Simon, L., Mallya, A., Bansal, A., , Gupta, G.: Coinductive logic programming, *22nd Int. Conf. Logic Programming (ICLP 2006)* (S. Etalle, M. Truszczyński, Eds.), 4079, Springer, August 2006.

[40] Simon, L., Mallya, A., Bansal, A., , Gupta, G.: Co-logic programming: Extending logic programming with coinduction, *34th Int. Colloq. Automata, Languages and Programming (ICALP 2007)* (L. Arge, C. Cachin, T. Jurdzinski, A. Tarlecki, Eds.), 4596, Springer, July 2007.

[41] Sperber, M., Thiemann, P.: ML and the Address Operator, *1998 ACM SIGPLAN Workshop on ML*, September 1998.

[42] Syme, D.: Initializing Mutually Referential Abstract Objects: The Value Recursion Challenge, *Proc. ACM-SIGPLAN Workshop on ML (2005)*, Elsevier, March 2006.

[43] Taylor, P.: *Practical Foundations of Mathematics*, Number 59 in Cambridge Studies in Advanced Mathematics, Cambridge University Press, 1999.

[44] Turbak, F. A., Wells, J. B.: Cycle Therapy: A Prescription for Fold and Unfold on Regular Trees, *PPDP*, ACM, 2001, ISBN 1-58113-388-X.

[45] y Widemann, B. T.: Coalgebraic Semantics of Recursion on Circular Data Structures, *CALCO Young Researchers Workshop (CALCO-jnr 2011)* (C. Cirstea, M. Seisenberger, T. Wilkinson, Eds.), August 2011.

[46] Wikipedia: p-adic numbers, `http://en.wikipedia.org/w/index.php?title=P-adic_number&oldid=553107165`, 2012.

[47] Wright, J. B., Thatcher, J. W., Wagner, E. G., Goguen, J. A.: Rational Algebraic Theories and Fixed-Point Solutions, *FOCS*, IEEE Computer Society, 1976.

[48] Zucca, E., Ancona, D.: Safe Corecursion in coFJ, *FTfJP'012 - Formal Techniques for Java-like Programs. 2013*, 2013.