Fundamenta Informaticae 150 (2017) 347–377
DOI 10.3233/FI-2017-1473
IOS Press

CoCaml: Functional Programming with Regular Coinductive Types

Jean-Baptiste Jeannin
Samsung Research America, USA
jb.jeannin@samsung.com

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

Alexandra Silva[*]
Department of Computer Science
University College London, UK
alexandra.silva@ucl.ac.uk

Abstract. Functional languages offer a high level of abstraction, which results in programs that are elegant and easy to understand. 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 are 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: Functional programming, coinductive types, recursive types, coalgebra

[*]Address for correspondence: Department of Computer Science, University College London, UK

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 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, hence theoretical developments have not yet fully found their place in mainstream implementations.

One possible approach to infinite coinductive datatypes, as embodied for example in lazy languages such as Haskell, treats infinite datatypes as *potential* objects, constructing only as much of them as needed. At any point during a computation, only a finite portion of the infinite object has ever been constructed. This approach requires a lazy evaluation strategy and does not fit well with call-by-value languages such as OCaml, which require arguments to be fully evaluated before the function is applied.

However, some eager call-by-value languages such as OCaml do have the ability to create certain infinite data objects, namely the *regular* ones; for example, infinite but ultimately periodic lists or infinite trees with finitely many subtrees up to isomorphism. The term *regular* refers to the fact that the object, though infinite, has a finite representation in the machine's memory. Typically this representation is a finite coalgebra, the represented object is an element of a final coalgebra, and the representation map is the unique coalgebra homomorphism from the former to the latter. This class of regular or rational coinductive types has been widely studied in the literature.

One might wonder whether the restriction to regular coinductive types might make computing with coinductive types 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 dataflow analysis in compiler construction. There are many interesting and useful functions that are computable for regular coinductive types that are not computable on general coinductive types; for example, the set of elements of an infinite ultimately periodic list, or the set of free variables of an infinite regular λ-term. From a theoretical perspective, they also offer many challenges in devising sound definitions and proof principles.

This paper presents a full implementation of new language constructs for computing with regular coinductive datatypes. One can already define regular coinductive types in OCaml, but the means to compute with them are limited. The usual recursive definitions that work on inductive (well-founded) types typically do not halt or provide the wrong solution when applied to infinite coinductive types. We do not change the way the datatypes are defined. Instead, we provide constructs that allow the programmer to specify how to solve equations resulting from recursive definitions applied to regular coinductive objects.

Let us provide some motivation using an example of a function over one of the simplest coinductive datatypes: finite and infinite lists. The infinite regular elements of this datatype are the eventually periodic lists. The built-in type `'a list` of OCaml consists of all finite and infinite lists of elements of type `'a`. The type definition consists 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 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 list of ones `[1; 1; 1; ...]` and the second the infinite alternating sequence `[1; 2; 1; 2; ...]`.

Although the **`let rec`** construct allows us to specify regular infinite lists, further investigation reveals a major shortcoming. Suppose we wanted to define a function that, given an infinite list, returns the set of its elements. For the lists `ones` and `alt`, we would like the function to 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 elements l = match l with
| [] -> []
| h :: t -> insert h (elements t)
```

where `insert` adds an element to a set, as represented say by a finite list without duplicates. Unfortunately, this function, applied to inputs `ones` and `alt`, will not halt in OCaml, even though it is clear in each case what the answer should be.

The 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 [1–5]. In [5], 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. 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 by a function application to a regular coinductive argument. 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; and `gaussian`, which solves a system of linear equations using gaussian elimination. 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 [1–4, 6–15], 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 addressed. 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 discuss the details behind the implementation. We discuss related work in §7, and in §8 we conclude and suggest some directions for future work.

2. Overview

Let us go back to the `elements` example from §1 and explain the steps necessary to obtain the desired solution. Note that when applied to `alt`, the program `elements` is essentially trying to compute a solution to a pair of equations

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

These equations are circular: `elements(alt)` depends on `elements(2::alt)` in the first equation and `elements(2::alt)` depends on `elements(alt)` in the second equation. This circularity would cause nontermination of the `elements` program under the usual semantics of recursive functions, because the data object `alt` is not well founded.

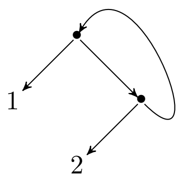
The issue is that the standard semantics of recursive functions gives the least solution in the flat Scott domain with bottom element $\bot$ representing nontermination. In this case, the least solution is $\bot$. However, the desired solution is the least solution in a different complete partial order, namely $\mathcal{P}(\mathbb{Z})$ ordered by set inclusion with bottom element $\varnothing$, where $\mathcal{P}(\mathbb{Z})$ denotes the powerset of the integers.

The main feature of our proposed solution is that we provide a mechanism for the programmer to specify an alternative method of solving the equations generated by a function call on a coinductive data object. For instance, in CoCaml, we could write:

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

The construct **`corec`** with the parameter `iterator []` specifies to the compiler that the equations above should be solved using the `iterator` solver—in this case a least fixpoint computation—

starting with the initial element `[]`. The infinite list `alt` can abstractly be viewed as the circular structure



Our extended compiler will generate two equations

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

one for each internal node of the circular structure (nodes are given fresh names), then solve them using the specified solver `iterator` starting from `[]`, which will produce the intended set $\{1, 2\}$.

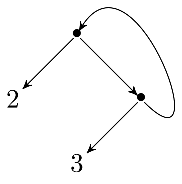
As another example in which a different type of solver is required, consider the `map` function, which applies a given function to every element of a given list. Again, the obvious definition, when applied to a circular structure, will not halt in OCaml. In CoCaml, we can specify that we want to get 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) = 2 :: (map(y))
map(y) = 3 :: (map(x))
```

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



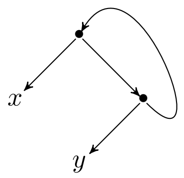
Another motivating example, which we borrow from [5], is the set of free variables of an infinitary λ-term (i.e., a λ-coterm). For ordinary well-founded λ-terms, the following definition works:

```
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 a 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 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 many of them, such as the set of free variables of an infinitary λ-term or the set of all elements of an infinite list, cannot. If one implements the free variables example in Haskell, then one cannot display the resulting set when the function is applied to a regular infinite term. One can still query membership, such as whether variable x is in `fv t`; if the answer is positive, then Haskell will be able to return it, however negative answers will be lost in an infinite search. Also note that even in cases where computation by lazy evaluation can be done and the regularity of the argument is preserved, this cannot be observed; only finite parts of the infinite unfolding of the term can be observed.

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 *bisimilar*: the two finite representations unfold to the same infinite object, or more accurately, have the same image in the final coalgebra.

The new **`corec`**`[solver]` construct has the advantage that no change is needed in the specification of the datatype (as in e.g. [4]) or in the usual definition of functions. Cycles in the input are detected automatically, enabling a built-in bisimilarity test, described in §4.2. 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 users with the means to define their 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 [16]. 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, 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 (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 [5] that this is not only useful, but feasible. In [5], 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 [5]. More importantly, we implement solvers in a much simpler and more elegant manner, and we provide users with the ability to specify their 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 references, making construction and manipulation of coalgebraic data easy.

When introducing mutable variables, particular care must be taken to handle type checking and type inference correctly, especially regarding polymorphism. Garrigue [17] proposes solutions in the context of references in ML, which we can adapt to mutable variables.

3.2. Capsule semantics

Our implementation is based on *capsule semantics* [18], 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

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

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.

Capsules allow rebinding of variables using the rebinding operator `x := v`. Here `x` is a variable that is in scope and `v` is a value. This should not be confused with the construct of the same name in OCaml, which is used exclusively to update references. In OCaml, the imperative statement `x := v` stores the new value `v` in the heap location to which the variable `x` is bound, but the binding of `x` remains intact; whereas in CoCaml, any variable of any type, if it is in scope, can be rebound to a new value of the same type. This construct, besides allowing CoCaml to model the assignment operator of imperative programs, also allows coinductive types and recursive functions to be constructed in a uniform way.

To construct regular coinductive types and recursive functions, we make use of 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 right-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, also known as *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 [13, 19–21]. Once `x` is rebound to a non-<> value, it can be deconstructed after looking it up in the environment. Note also that the value <> is not accessible to the programmer, as there is no syntax for it. It is only used internally in the implementation of `let rec` as described above.

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 §4.2). For this reason, coalgebraic types are not really the same as the circular data structures as studied in [19–22]. The example above shows that we now allow the runtime 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

[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.

styles in a common framework, which is one of the main motivating factors behind capsules. For a basic introduction to capsule semantics, see [18], and for a full account of capsule semantics in the presence of coalgebraic types, see [23].

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 [24].

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

$$\begin{tikzcd} C \arrow[r, "h"] \arrow[d, "\gamma"'] & A \\ FC \arrow[r, "Fh"] & FA \arrow[u, "\alpha"'] \end{tikzcd} \tag{1}$$

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 inductive datatypes (in other words, datatypes defined as the initial algebra of a functor F) 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 to another F-algebra A. Another (dual) setting in which it is easy to obtain h uniquely is when the codomain A is a final coalgebra.

This general idea has been well studied [1,6,9,25,26]. 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 [13] on coalgebraic semantics of recursion and cycle detection algorithms, the work on coinductive logic programming [2,11,27], and the work on coinductive featherweight Java [3,8], which address many of the same issues but in the context of logic and object-oriented programming.

As mentioned, 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. This was the object of study in the 2006 paper of Adamek, Milius and Velebil [1] and in the recent papers of Jeannin, Kozen and Silva [5,28].

The example from §2 involving free variables fits this scheme precisely. Instantiating the diagram (1) yields:

$$\begin{tikzcd}
\texttt{Term} \arrow[r, "\texttt{fv}"] \arrow[d, "\gamma"'] & \mathcal{P}(\texttt{Var}) \\
F(\texttt{Term}) \arrow[r, "\mathsf{id}_{\texttt{Var}} + \texttt{fv}^2 + \mathsf{id}_{\texttt{Var}} \times \texttt{fv}"'] & F(\mathcal{P}(\texttt{Var})) \arrow[u, "\alpha"']
\end{tikzcd}$$

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 $f(a)$, 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, which 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 result of a call of f on some input a_i. Its right-hand side is a partially evaluated abstract syntax tree, an expression of the language that 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 the function f. This computation finishes 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 necessarily well-founded. But because a_0 has a finite representation, the set of possible such calls is finite, say for example on $a_1, \ldots, a_n$.

To handle 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. This results in a finite set of equations.

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

4.2. Equality of coinductive data

Equality for coinductive data values is synonymous with *bisimilarity*; that is, two data values are considered equal if they are observationally equivalent. As regular coinductive values have finite representations, equality is computable in finite time, unlike with lazy approaches. In CoCaml, this form of equality is built in and can be tested with the operator =.

Unfortunately, OCaml's documentation tells us that "equality between cyclic data structures may not terminate." In practice, an OCaml equality test with = returns `false` if it finds a difference in finite time, but loops forever 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 *)
```

In CoCaml, an equality test with = halts and returns `true` if the two arguments are bisimilar and `false` if not. In the example above, the calls `zeros = zeros` and `zeros = zeros2` both 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.

Equality of coinductive data values plays a central role in the generation of 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 this, 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.

In CoCaml, coinductive data values and recursive functions are represented internally as capsules, thus it suffices to implement 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 $\mathsf{Cap}^2 = \mathsf{Cap} \times \mathsf{Cap}$, the set of pairs of capsules. The codomain is the two-element Boolean algebra $\nvDash$. The diagram (1) is instantiated to

$$\begin{tikzcd}
\mathsf{Cap}^2 \arrow[r, "h"] \arrow[d, "\gamma"'] & \nvDash \\
\nvDash + \mathsf{Cap}^2 + \mathsf{list}\ (\mathsf{Cap}^2) \arrow[r, "\mathsf{id}_{\nvDash} + h + \mathsf{map}\ h"'] & \nvDash + \nvDash + \mathsf{list}\ \nvDash \arrow[u, "\alpha"']
\end{tikzcd}$$

where the functor is $FX = \nvDash + X + \mathsf{list}\ X$. 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 the ones in which equality must be recursively determined. If they are both literal integers or some other base type, 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.

A naive implementation of this algorithm is quadratic, as it compares all pairs of states in the capsules in the worst case. However, it turns out that we can regard the capsules as deterministic finite automata, and the capsules are equal if and only if the corresponding automata are equivalent. There is a known $O(n\alpha(n))$ algorithm of Hopcroft and Karp [29] for this task, where α is the inverse of the Ackermann function. Briefly, the algorithm maintains a partition of elements that are forced to be bisimilar if the original pair elements are to be bisimilar. Starting with just the initial pair, bisimilarity constraints are propagated forward according to the structure of the data, causing partition elements to be merged. If ever two non-matching elements are forced to be merged, then the algorithm halts and returns `false`, declaring the original pair of elements unequal. If no such conflicts are discovered after processing all reachable states, the algorithm halts and returns `true`, declaring the original pair of elements equal. The complexity stems from the use of the so-called *union-find* data structure for maintaining disjoint sets, which has $O(m\alpha(n))$ amortized complexity for m union and find operations over a set of size n.

4.3. 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. Moreover, 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 [5, 30], 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.

4.4. 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.

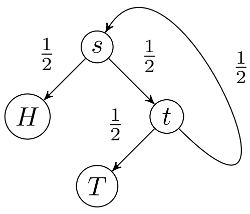
Example. The map function on lists takes a function f and a list l, applies f on every element h of l, and returns the list of the results f h. The constructor solver can be used to create a map functions that works on all lists, finite or infinite.

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

4.5. 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.

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

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

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

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] pr_heads t = match t with
| Heads -> 1.
| Tails -> 0.
| Flip(p,t1,t2) -> p *. pr_heads t1 +. (1. -. p) *. pr_heads t2
```

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, programmers sometimes need to define their own solvers. This can be done by defining three functions `unk`, `fresh` and `solve` with the following signatures:

```
type var
type expr
type t
val unk : var -> expr
val fresh : unit -> var
val solve : var -> (var * expr) list -> t
```

The 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. The type `expr` acts as a small abstract syntax tree of equations. The user can manipulate and solve the equations without access to the full CoCaml abstract syntax tree.

The 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, the type `var` can simply be `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`. The function `unk` does this. Typically `expr` is a sum type that contains a special case `Unknown of var`, and the injection `unk` is just

```
let unk 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, thus is represented as a pair of type `var * expr`. Given an element x of type `var` and 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 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
```

The functions `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 `Mul` to have arguments `float*expr`, which would automatically keep the equations linear. We write instead

`Mul of expr*expr` and check linearity dynamically. The declaration of the function `pr_heads` becomes

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

The right-hand side is slightly different and less clear than in the original definition. Instead of working with the abstract syntax of the whole language, the programmer defines a specialized abstract syntax 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 [31–33].

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 illustrate the use of our main solvers through several examples on finite and infinite lists, one of the simplest examples of coinductive types. Through these examples, we show how easy it is to create recursive functions on finite and infinite regular lists, as the process is very close to creating recursive functions on inductive datatypes.

5.1.1. 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` and `forall`

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

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

The function `forall` is the same, but with `||` replaced by `&&` and `false` by `true`.

5.1.3. Pretty printing

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 a = 1 :: 2 :: a;;
val a : int list = [1; 2; 1; 2; 1; 2; 1; 2; ...]
```

This is not very satisfying. Often it may appear as if some pattern is repeating, but what if for instance a 3 appears in 50th position and is not printed? A better solution might be to print the initial non-periodic prefix followed by the periodic part as a starred expression. For instance, the list `[1; 2]` might be printed as `[1; 2]`, the list `a` above as `[(1; 2)*]`, and the list `1 :: 3 :: a` as `[1; 3; (1; 2)*]`. This can be achieved by creating a function that partitions a given list into an initial non-periodic prefix and, if infinite, a periodic suffix. We can use the built-in observational equality test = for this task. We simply keep track of all suffixes, halting as soon as a suffix is repeated or the end of the list is reached.

```
let rec first_recurrence a s l =
  if mem l s then (rev a,s,l)
  else match l with
  | [] -> (rev a,s,l)
  | h :: t -> first_recurrence (h :: a) (l :: s) t

let separate l =
  match first_recurrence [] [] l with (a,s,l) ->
  let n = position l s in
  let m = length a - n in
  let s = take a m in
  let t = take (snd s) n in
  (fst s, fst t)
```

Here `mem` tests membership in a list using the equality predicate `=`, `position` calculates the position of an element in a list, and `take a n` takes the first `n` elements of list `a`. The call `separate a` will produce the pair `[], [1; 2]`, the call `separate (1 :: 3 :: a)` will produce the pair `[1; 3], [1; 2]`, and the call `separate [1; 2; 3]` will produce the pair `[1; 2; 3], []`.

Note that this computation does not break representation independence. The output partition is the minimal one and is uniquely determined, irrespective of the representation of the input list.

5.1.4. Infinitely-often membership

The `separate` function defined in §5.1.3 can also be used to determine the set of elements occurring infinitely often in a given list. This is also not computable lazily. If a call to `separate` returns the finite non-periodic prefix ℓ_1 and the periodic part ℓ_2, then the set of elements occurring infinitely often are exactly those elements of ℓ_2, and the set of elements occurring only finite often are those elements of ℓ_1 that do not occur in ℓ_2.

5.1.5. The curious case of filtering

Given a Boolean-valued 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, considering each call to its first argument `f` to be $O(1)$.

5.1.6. 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.

5.2. A library for p-adic numbers

The p-adic numbers are a well-studied mathematical structure with applications in several areas of mathematics and computer science [34–36]. In this section we present a library for rational p-adic numbers and operations on them.

5.2.1. The p-adic numbers

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 as a formal sum $\sum_{i=k}^{\infty} d_i p^i$, 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, [])
| _ -> 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 implementation 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. 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 $k \geq 0$, $x_k = d_k + px_{k+1}$. 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 $x_k = d_k + px_{k+1}$ 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 fractional 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 surprisingly easy. We can use a slight adaptation of the primary school algorithm of adding digit by digit with carries. A carry might come from adding the fractional parts, so the algorithm really takes three arguments, the two p-adic integers 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 rational 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. Other examples

Besides the examples presented here, we have implemented all the examples that Jeannin, Kozen and Silva collected in [5] to CoCaml. Among those, substitution in an 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 programs 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 ordinary deterministic and nondeterministic finite automata.

To complement the example of §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, attempting to generate Fig. 1(d).

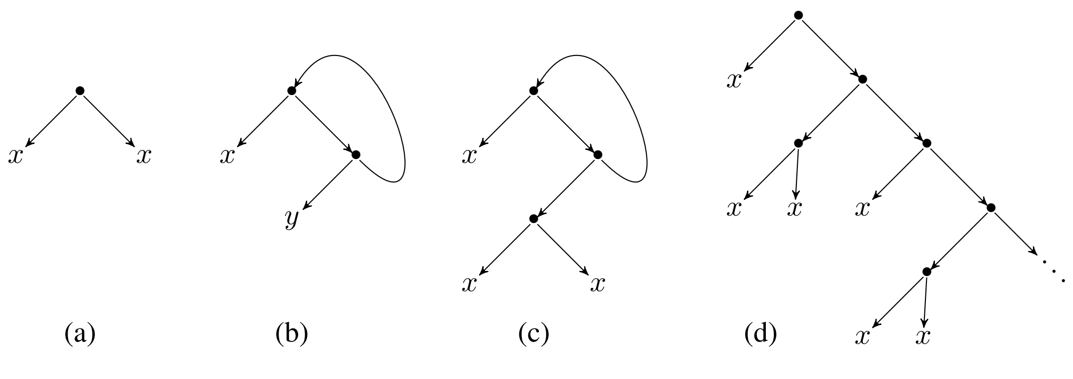


Figure 1. A substitution example.

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.

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 `solver`.

Equations can be correctly generated only if all recursive calls are applied to an argument and none of them are nested. The argument must be explicit, thus examples such as

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

are disallowed. Right-hand sides may 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 take 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 the way 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 the 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 for efficiency reasons: if the first argument of `&&` evaluates to `false`, then it should return `false`; and similarly for `||` and `true`. This laziness, also implemented in CoCaml, only changes the semantics if the second argument produces an error or does not finish. However, it is also possible that the first argument only evaluates partially. In that case, we also evaluate the second argument, so that we can generate unknown variables and equations for all recursive calls appearing in the body of the function. Another choice would not make sense: if we chose to not evaluate the second argument at first, and if its evaluation became necessary later on, then we would have no way of evaluating the recursive calls of the second argument after the equation generation.

7. Related work

Syme [21] 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 [5] and in the work of Ancona which we discuss below.

Sperber and Thiemann [20] 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 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 addresses the same issues as Felleisen's and Hieb's theories of syntactic state [37], but capsules are considerably simpler. A major advantage is the elimination of the explicit context present in [37].

Hirschowitz, Leroy, and Wells [19] 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 [13], 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 "front-end language" constructs as an important problem for future work, which is one of the issues we have addressed here.

Recently, copatterns [38, 39] 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 [40], where the `cyclist` library can be found. This library provides some functions on infinite lists in OCaml. However, it is limited to lists and does not handle any other coinductive type. Another relevant paper where functions on lists in an ML-like language are discussed is [41]. There is also work in Scheme [42] which defines 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 addressed. Coinductive logic programming (coLP) [11, 27] has been recently introduced as a step 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 define corecursive FeatherWeight Java (coFJ) [8], which extends FeatherWeight Java with language support for cyclic structures. In [3] they provide a translation from coFJ into coLP, clarifying the connections between the two frameworks and providing an effective implementation of coFJ. In [15], they define a type system for coFJ that allows the user to specify that certain methods are not allowed to return the value *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 [2] 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. For example, in Coq one can define coinductive datatypes similar to the ones found in OCaml, along with functions and proofs on them [43,44]. 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 [43]. This is sometimes restrictive, and less restrictive conditions have been developed focusing on programs in Coq [44,45] and Agda [46] or on proofs in Coq [47,48].

Work on cyclic structures in lazy languages can be found in e.g. [4, 10, 12, 49]. In these works, explicit modeling of back pointers and visited nodes is used, requiring the use of nested datatypes. No new programming constructs are proposed. In our work, we do not touch the datatype definition: the cyclic structure of an object is detected automatically.

The closest to this work in the context of OCaml is our earlier work [5], in which we discuss at length what is needed to extend OCaml with language support to define functions on regular coinductive types. A mock-up implementation in an OCaml-like language is provided. However, our earlier proposal to specify alternative semantics (solution methods) is not generic and there is no support for user-defined solvers. In this paper, we have given a full implementation and improved on our earlier results by showing how an alternative semantics, provided by equation solvers, can be given in a lightweight, elegant fashion. We have implemented several generic solvers, which are versatile enough to cover a wide range of examples, and give users a mechanism to define their own solvers. All the examples discussed in [5] 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 through 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 means to define custom 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 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 [33], 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 to connect our work with recent work on coinduction in program verification [50].

Acknowledgments

We extend our sincere 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; to Hersh Mehta for help with the implementation; and to the anonymous referees for their many insightful suggestions.

References

[1] Adámek J, Milius S, Velebil J. Elgot Algebras, Log. Methods Comput. Sci., 2006;2(5:4)1–31. doi:10.2168/LMCS-2(5:4)2006.

[2] Ancona D. Regular corecursion in Prolog, SAC (S. Ossowski, P. Lecca, Eds.), ACM, 2012, ISBN 978-1-4503-0857-1. doi:10.1145/2245276.2232088.

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

[4] 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.

[5] Jeannin JB, 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. doi:10.1007/978-3-642-37036-6_4.

[6] Adámek J, Lücke D, Milius S. Recursive Coalgebras of Finitary Functors, Theoretical Informatics and Applications, 2007;41:447–462. doi:10.1051/ita:2007028.

[7] Adámek, J., Milius, S., Velebil, J.: Iterative algebras at work, Mathematical. Structures in Comp. Sci., December 2006;16(6):1085–1131. doi:10.1017/S0960129506005706.

[8] Ancona D, Zucca E. Corecursive Featherweight Java, FTfJP'012 - Formal Techniques for Java-like Programs, 2012. doi:10.1145/2318202.2318205.

[9] 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. doi:10.1007/978-3-642-10452-7_7.

[10] 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. doi:10.1145/237721.237792.

[11] 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. doi:10.1007/978-3-540-73420-8_42.

[12] Turbak FA, Wells JB. Cycle Therapy: A Prescription for Fold and Unfold on Regular Trees, PPDP, ACM, 2001, ISBN: 1-58113-388-X. doi:10.1145/773184.773200.

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

[14] Wright JB, Thatcher JW, Wagner EG, Goguen JA. Rational Algebraic Theories and Fixed-Point Solutions, FOCS, IEEE Computer Society, 1976. doi:10.1109/SFCS.1976.24.

[15] Zucca E, Ancona D. Safe Corecursion in coFJ, FTfJP'012 - Formal Techniques for Java-like Programs. 2013, 2013. doi:10.1145/2489804.2489807.

[16] Goguen JA, Thatcher JW. Initial Algebra Semantics, 15th Symp. Switching and Automata Theory, IEEE, 1974. doi:10.1109/SWAT.1974.13.

[17] Garrigue J. Relaxing the Value Restriction, Proceedings of the 7th International Symposium on Functional and Logic Programming, FLOPS 2004, Nara, Japan, April 7-9, 2004. (Y. Kameyama, P. J. Stuckey, Eds.), Springer, Berlin, Heidelberg, 2004, ISBN: 978-3-540-24754-8. doi:10.1007/978-3-540-24754-8_15.

[18] Jeannin JB, Kozen D. Computing with Capsules, J. Automata, Languages and Combinatorics, 2012;17(2–4):185–204.

[19] Hirschowitz T, Leroy X, Wells JB. Compilation of extended recursion in call-by-value functional languages, PPDP 2003, 2003. doi:10.1145/888251.888267.

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

[21] Syme D. Initializing Mutually Referential Abstract Objects: The Value Recursion Challenge, Proc. ACM-SIGPLAN Workshop on ML (2005), Elsevier, March 2006. doi:10.1016/j.entcs.2005.11.038.

[22] 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.

[23] 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. doi:10.1016/j.entcs.2012.08.003.

[24] Jeannin JB. 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. doi:10.1016/j.entcs.2011.09.022.

[25] Eppendahl A. Coalgebra-to-Algebra Morphisms, Electronic Notes in Theoretical Computer Science, 29, 1999. doi:10.1016/S1571-0661(05)80305-6.

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

[27] 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. doi:10.1007/11799573_25.

[28] Jeannin JB, Kozen D, Silva A. Well-Founded Coalgebras, Revisited, Math. Structures in Computer Science, FirstView:1-21, February 2016. doi:10.1017/S0960129515000481.

[29] Hopcroft JE, Karp RM. A linear algorithm for testing equivalence of finite automata, Technical report, Cornell University, 1971.

[30] Pottier F. Lazy Least Fixed Points in ML, URL `pauillac.inria.fr/~fpottier/publis/fpottier-fix.pdf`.

[31] Capretta V. Coalgebras in functional programming and type theory, Theor. Comput. Sci., 2011; 412(38):5006–5024. doi:10.1016/j.tcs.2011.04.024.

[32] Silva A, Rutten JJ. Behavioural Differential Equations and Coinduction for Binary Trees, WoLLIC (D. Leivant, R. J. G. B. de Queiroz, Eds.), 4576, Springer, 2007, p. 322–336. ISBN: 978-3-540-73443-7. doi:10.1007/978-3-540-73445-1_23.

[33] Silva A, Rutten JJ. A coinductive calculus of binary trees, Inf. Comput., 2010;208(5):578–593. doi:10.1016/j.ic.2008.08.006.

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

[35] URL `https://forge.ocamlcore.org/projects/ocaml-cyclist/`, June 2010.

Hehner ECR, Horspool RNS. A new representation of the rational numbers for fast easy arithmetic, SIAM J. Comput., 1979;8(2):124–134. doi:10.1137/0208011.

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

[37] Felleisen M, Hieb R. The Revised Report on the Syntactic Theories of Sequential Control and State, Theoretical Computer Science, 1992;103:235–271. doi:10.1016/0304-3975(92)90014-7.

[38] Abel A, Pientka B. Wellfounded recursion with copatterns: a unified approach to termination and productivity, ICFP, 2013. doi:10.1145/2500365.2500591.

[39] Abel A, Pientka B, Thibodeau D, Setzer A. Copatterns: programming infinite structures by observations, POPL, 2013. doi:10.1145/2429069.2429075.

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

[41] Caspi P, Pouzet M. A Co-iterative Characterization of Synchronous Stream Functions, Electr. Notes Theor. Comput. Sci., 1998;11:1–21. doi:10.1016/S1571-0661(04)00050-7.

[42] Adams MD, Dybvig RK. Efficient nondestructive equality checking for trees and graphs, Proc. 13 ACM SIGPLAN Int. Conf. Functional Programming, 2008. doi:10.1145/1411204.1411230.

[43] Coquand T. Infinite Objects in Type Theory, TYPES, 1993. doi:10.1007/3-540-58085-9_72.

[44] Giménez E. Structural Recursive Definitions in Type Theory, ICALP, 1998. doi:10.1007/BFb0055070.

[45] Bertot Y, Komendantskaya E. Inductive and Coinductive Components of Corecursive Functions in Coq, Electr. Notes Theor. Comput. Sci., 2008;203(5):25–47. doi:10.1016/j.entcs.2008.05.018.

[46] Danielsson NA. Beating the Productivity Checker Using Embedded Languages, PAR, 2010. doi:10.4204/EPTCS.43.3.

[47] Hur CK, Neis G, Dreyer D, Vafeiadis V. The power of parameterization in coinductive proof, POPL, 2013. doi:10.1145/2429069.2429093.

[48] Niqui M. Coalgebraic Reasoning in Coq: Bisimulation and the lambda-Coiteration Scheme, TYPES, 2008. doi:10.1007/978-3-642-02444-3_17.

[49] Oliveira, Bruno CdS, Cook WR. Functional programming with structured graphs, ICFP (P. Thiemann, R. B. Findler, Eds.), ACM, 2012, p. 77–88. ISBN 978-1-4503-1054-3. doi:10.1145/2364527.2364541.

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