CoCaml: Programming with Coinductive Types

Jean-Baptiste Jeannin* Dexter Kozen* Alexandra Silva†

December 27, 2012

Abstract

We present CoCaml, a functional programming language extending OCaml, which allows us to define functions on coinductive datatypes parameterized by an equation solver. We provide numerous examples that attest to the usefulness of the new programming constructs, including operations on infinite lists, infinitary λ-terms and p-adic numbers.

keywords: coalgebraic types, functional programming, recursion

1 Introduction

Infinite datatypes and functions on infinite datatypes offer interesting challenges in the design of programming languages. While most programmers feel comfortable with inductive datatypes and functions on them, coinductive datatypes are often considered difficult to handle, and many programming languages do not even offer the constructs to define them.

OCaml offers the possibility of defining coinductive datatypes, but the means to define recursive functions on them are limited. Often the obvious definitions do not halt or provide the wrong solution.

Let us provide some motivation using an example of a function over what has been referred to as the simplest coinductive datatype: infinite lists. The type of finite and infinite integer lists can be specified in OCaml by

```
type list = N | C of int * list
```

Infinite lists can now be defined coinductively using the `let rec` construct:

```
let rec ones = C(1, ones)
let rec alt = C(1, C(2, alt))
```

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

*Computer Science Department, Cornell University, Ithaca, New York 14853–7501, USA. Email: {jeannin, kozen}@cs.cornell.edu

†Institute for Computing and Information Sciences, Radboud University Nijmegen, Postbus 9010, 6500 GL Nijmegen, The Netherlands. Email: alexandra@cs.ru.nl

The `let rec` construct allows us to build only *regular* lists, that is, those that are ultimately periodic. Such lists always have a finite representation in memory. The coinductive elements we consider are always regular, i.e., they have a finite but possibly cyclic representation. 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, for example free variables, cannot.

Although the `let rec` construct allows us to specify (finite representations of) infinite structures, 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. One would like to write a function definition using the obvious equations which pattern-match on the two constructors of the `list` datatype:

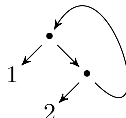
```
let set l = match l with
  | N -> N
  | C(h, t) -> insert h (set t)
```

where `insert` inserts an element in a set, represented by a finite list without duplicates. However, this function will not halt in OCaml on the lists `ones` and `alt`, even though it is clear what the answers should be. Note that this is not a corecursive definition, 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 $\perp$ representing nontermination, whereas we would like the least solution in a different CPO, namely $(\mathcal{P}(\mathbb{Z}), \subseteq)$ with bottom element $\varnothing$.

In this paper, we present CoCaml, an extension of OCaml in which functions defined by equations, like the one above, can be supplied with an extra parameter, namely a solver for the given equations. For instance, the example above would be almost the same in CoCaml:

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

The construct `corec` with the parameter `iterator(N)` 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 `N`. For the infinite list `alt`, which can abstractly be thought of as the circular structure



the compiler will generate two equations:

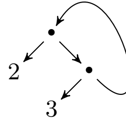
```
set(x) = insert 1 (set(y))
set(y) = insert 2 (set(x))
```

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

For another example, 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; more abstractly, we actually compute the solution in the final coalgebra. Again, the definition looks very much like the standard one:

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

As desired, applications to circular structures halt and produce the expected result. For instance, `map plusOne alt` will produce the infinite list $2, 3, 2, 3, \ldots$ as represented by the circular structure



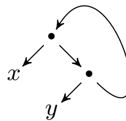
Another motivating example from [11] is the set of free variables of an infinitary λ-term (λ-coterm). For ordinary well-founded λ-terms, the following definition works:

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

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

However, if we call the function on an infinite 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(N)` solver as in the example above involving elements of an infinite regular list.

The paper is organized as follows. In §2, we describe coinductive types in the context of a functional language and the theoretical framework on which our new language constructs are based. We also describe *capsule semantics*, a heap-free mathematical semantics for higher order functional and imperative programs, which provides the foundation for our implementation. In §3, we discuss the subtleties of equality on cyclic data structures and show how to implement it to ensure termination. In §4, we dive into the details behind the implementation. In §5, we give several detailed examples illustrating the use of the new constructs, including functions on finite and infinite lists, p-adic numbers, and infinitary λ-terms. In §6, we mention limitations of the framework and direction for future work. In §7, we discuss relevant related work, and in §8 we give concluding remarks.

2 Framework

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.

2.1 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 [7]. 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.

A more subtle distinction is that 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 ...
```

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 [11] that this is not only useful, but feasible. In [11], 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 realistic implementation in an OCaml-like language. We also give several new examples of its usefulness in addition to the examples of [11].

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 programmatically. 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.

2.2 Theoretical Foundations

A very general picture of the situation is given by the commuting diagram

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

describing how a recursive function $h : C \to A$ is defined. Here F is a functor that determines the structure of the base cases and the 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. The function $\alpha : FA \to A$ takes the return values from the recursive calls and assembles them into the return value $h(x)$.

Ordinary recursively defined functions on well-founded datatypes fall into this framework. For example, the factorial function

```
let rec fact = function
  | 0 -> 1
  | n -> n * fact(n-1)
```

has the diagram

$$\begin{array}{ccc} \mathbb{N} & \xrightarrow{\texttt{fact}} & \mathbb{N} \\ {\scriptstyle\gamma}\downarrow & & \uparrow{\scriptstyle\alpha} \\ \mathbb{1} + \mathbb{N} \times \mathbb{N} & \xrightarrow[\mathsf{id}_{\mathbb{1}} + \mathsf{id}_{\mathbb{N}} \times \texttt{fact}]{} & \mathbb{1} + \mathbb{N} \times \mathbb{N} \end{array}$$

where the functor is $FX = \mathbb{1} + \mathbb{N} \times X$ and γ and α are given by:

$$\begin{aligned} \gamma(0) &= \iota_0() & \qquad \alpha(\iota_0()) &= 1 \\ \gamma(n+1) &= \iota_1(n+1, n) & \qquad \alpha(\iota_1(c, d)) &= cd \end{aligned}$$

where the ι_0 and ι_1 are injectors into the coproduct. The fact that there is one recursive call is reflected in the functor by the single X occurring on the right-hand side. The destructor γ determines whether the argument is a base case, and if not, prepares the recursive call. The constructor α combines the result of the recursive call with the input value by multiplication. In this case we have a unique solution, which is precisely the factorial function.

This general idea has been well studied [1, 2, 4, 6, 15]. 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.

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.

The free variables example from §1 fits this scheme with the diagram

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

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

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

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 complete 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. Our language allows the programmer to specify an alternative solution method in such cases.

2.3 Capsule Semantics

Our implementation is based on *capsule semantics* [10], a heap-free mathematical semantics for higher order functional and imperative programs. This semantics admits mutable let-bound variables a la Scheme. 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.)

In capsule semantics, 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 a uniform way via backpatching (aka 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. (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.) 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 [9, 13, 14, 17]. 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:

```
type t = C of t
let rec x = C(C(C(x))) in x := C(x)
```

Any cycle must always contain at least one such variable. Note that these two cyclic data objects actually represent the same infinite object, namely the infinite term `C(C(C(...`. Two elements of a coalgebraic type are considered equal iff they are bisimilar (see §3). For this reason, coalgebraic types are not really the same as the circular data structures as studied in [9, 13, 14].

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 [10], and for a full account of capsule semantics in the presence of coalgebraic types, see [12].

3 Corecursive Equality

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. Let us describe the 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 this equality is the set of pairs of capsules, defining the coalgebra $\mathsf{Cap}^2 = \mathsf{Cap} \times \mathsf{Cap}$. 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 $FX = 2 + X + \mathsf{list}\ X$, $\mathsf{list}\ X$ 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. If either one is a variable, it looks up its value in the corresponding environment. If they are both literal integers, it returns $\iota_1(\texttt{true})$ if they are equal and $\iota_1(\texttt{false})$ otherwise. 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 is the identity on the first two projections, and on $\iota_3(l)$ returns `true` if all the elements of l are true, `false` otherwise.

In [11], we gave an OCaml functor allowing us to generate h, given a solver and an equality on C. We use this functor. The solver must return `false` if an $\iota_1(\texttt{false})$ was ever encountered, `true` otherwise. It is a simple iteration over the equations.

The only remaining question is equality on C. We are asking whether a capsule has already been seen, and here we can revert to physical equality `==`. Because the pairs that form the capsules are destructed and reconstructed throughout the algorithm, equality needs to be:

```
let equal ((s1, o1), (t1, p2)) ((s2, o2), (t2, p2)) =
  s1 == s2 && o1 == o2 && t1 == t2 && p1 == p2
```

The generated function behaves as desired:

```
> zeros = ones;;
- : bool = false
> zeros = zeros;;
- : bool = true
> zeros = zeros2;;
- : bool = true
```

This is the last time we use the system described in [11]. In a sense, by allowing us to program equality, that system allows us to bootstrap CoCaml.

4 Implementation

When a function `f` is defined using the `corec` keyword, it is first typed as an ordinary recursive function, using the Hindley–Milner type inference algorithm, as is usual in ML. If this type check succeeds, `f` 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 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_0. Since our language is call-by-value, a_0 is first evaluated and bound in the current environment. We then proceed to generate the recursive equations that the value of $\mathtt{f}(\mathsf{a}_0)$ must satisfy.

4.1 Generating Equations

The equations for $\mathtt{f}(\mathtt{a}_0)$ will involve a finite number of recursive calls to `f` on the arguments $\mathtt{a}_0, \mathtt{a}_1, \ldots, \mathtt{a}_n$. To generate the equations, we first create variables $\mathtt{x}_0, \mathtt{x}_1, \ldots, \mathtt{x}_n$. These variables are unknowns denoting the values of $\mathtt{f}(\mathtt{a}_0), \mathtt{f}(\mathtt{a}_1), \ldots, \mathtt{f}(\mathtt{a}_n)$, respectively. When partially evaluating the body of `f` applied to some $\mathtt{a}_i$, we might encounter some calls to some $\mathtt{f}(\mathtt{a}_j)$, and we replace each such call by the variable $\mathtt{x}_j$. This gives a set of equations of the form $\mathtt{x}_i = \mathtt{e}_i$, where $\mathtt{e}_i$ is a partially evaluated expression involving the variables $\mathtt{x}_0, \mathtt{x}_1, \ldots, \mathtt{x}_n$. This system of equations is then passed to a solver (see §4.3).

Of course, the arguments $\mathtt{a}_0, \mathtt{a}_1, \ldots, \mathtt{a}_n$ are not known in advance, as we only know $\mathtt{a}_0$. Therefore, in the partial evaluation, a new $\mathtt{x}_i$ is generated whenever a new $\mathtt{a}_i$ is encountered. This is achieved by keeping track of all the $\mathtt{a}_i$ that have been seen so far, along with their associated unknowns $\mathtt{x}_i$.

4.2 Partial Evaluation

Partial evaluation is much like normal evaluation except when encountering a recursive call to `f`. When such a recursive call $\mathtt{f}(\mathtt{a}_j)$ is encountered, we have seen that its argument is evaluated, and the call is replaced by a variable $\mathtt{x}_j$ corresponding to $\mathtt{a}_j$. The variable $\mathtt{x}_j$ might be fresh if $\mathtt{a}_j$ had not been seen before, or it might be the one already associated with $\mathtt{a}_j$.

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 due to laziness, 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.

4.3 Solvers

A solver receives a set of equations. These equations have right-hand sides that are partially evaluated abstract syntax trees and contain unknowns. Depending on the solver chosen, it manipulates these abstract syntax trees to find a solution. We currently have five solvers implemented, of which four are quite versatile and can be used in many different applications. These solvers are described along with examples of their use in §5.

5 Examples and Solvers

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 one 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 coinductive datatypes, as the process is very close to creating recursive functions on inductive datatypes.

5.1.1 Test of Finiteness and the `appears` Solver

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
  | N -> true
  | C(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 `C(0, N)` will look like

```
is_finite(C(0, N)) = is_finite(N)
is_finite(N) = 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 is what the solver `solver(`e`)` does: it returns true if and only if the expression e appears on the right-hand side of one of the equations.

Our test of finiteness thus becomes:

```
let corec[appears(true)] is_finite l = match l with
  | N -> true
  | C(h, t) -> (is_finite t)
```

An alternative approach for this example would use the `iterator` solver of §5.1.3.

5.1.2 Mapping a Function and 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 element of a final coalgebra. 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, N -> N
  | f, C(h, t) -> C(f(h), map (f,t))
```

The `constructor` solver takes care of building the new data structure, even if it turns out to be circular. 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.

5.1.3 Sets of Elements and 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.

We can apply this to creating 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 α is the type of the elements, the codomain of `set` is the CPO $(\mathcal{P}(\alpha), \subseteq)$ with bottom element $\varnothing$. Restricted to subsets of the set of variables appearing in the list, it satisfies the ACC.

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

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

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 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 [11], but it can also be used in applications where the desired fixpoint is not necessarily the least.

5.1.4 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`. This can be simply programmed using either the `appears` or `iterator` solver:

```
let corec[appears(true)] exists arg = match arg with
  | f, N -> false
  | f, C(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.

5.1.5 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, N -> N
  | f, C(h, t) -> if f(h) then C(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 set `N` 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 `N` explicitly when needed:

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

5.1.6 Printing and 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 = C(1, ones);;
val ones : lis = C (1, C (1, C (1, C (1, ... ))))
```

This is not very satisfying. Often it may appear as if some pattern is repeating, but what if for example 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 `C(1, C(2, N))` might be printed as `12` and the list `C(1, C(2, ones))` would be printed as `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.

The function extracts the equations defining the list and passes them to the solver. The solver expects equations with right-hand sides that are injections of `I1` and `I2` and returns an injection of `I3` containing both lists.

```
type sep = I1 | I2 of int * sep | I3 of lis * lis'

let corec[separate] separate i = match i with
  | N -> I1
  | C(i, t) -> I2(i, separate t)
```

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.

However, this example is not completely satisfying. The function shown above seems to not do anything, and all the work is done in the solver. As such, the solver is quite ad hoc, which contrasts greatly with the solvers we have seen so far. Moreover, the type `sep` we have introduced exists merely to make the type checker happy. Conceptually, the solver takes a list as an argument and returns a pair of lists. This example shows the limits of the typing mechanism as applied to functions on coinductive data.

5.1.7 Other Examples

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 [5] for more details.

Other examples involving probabilistic protocols, finite automata, and abstract interpretation are given in [11].

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 [3, 16] 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 example 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 of 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 `appears` solver and compares two integer parts of p-adic numbers for equality:

```
let corec[appears(false)] equali_aux x = match x with
  | N, N -> true
  | C(h1, t1), C(h2, t2) -> h1 = h2 && equali_aux (t1, t2)
  | C(0, t1), N -> equali_aux (t1, N)
  | N, C(0, t2) -> equali_aux (t2, N)
  | _ -> false

let equali x = not (equali_aux x)
```

The two arguments are not equal if and only if `false` appears on the right-hand side of one of the equations. This is not the same as if `true` does not appear, thus the necessity of an auxiliary function.

Interestingly, comparing the fractional parts is almost the same code, with the `rec` keyword instead of the `corec` keyword, and no auxiliary function.

```
let rec equalf x = match x with
  | N, N -> true
  | C(h1, t1), C(h2, t2) -> h1 = h2 && equalf (t1, t2)
  | C(0, t1), N -> equalf (t1, N)
  | N, C(0, t2) -> equalf (t2, N)
  | _ -> false

let equal x1 x2 = match x1, x2 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, N) then N
  else match i with C(i, t) -> C(i, normalizei t)
```

The function `normalizei` only requires equality with zero (represented as `N`), 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 N
  else let d = euclid p a b in
  C(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 and the `gaussian` Solver

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 create a `gaussian` solver that solves equations when the right-hand sides are linear functions. Our Gaussian elimination is on floats and returns a float (thus an approximation), but we could as well have returned a fraction. The equations are created with the following function:

```
let corec[gaussian] to_floati i = match i with
  | N -> 0.
  | C(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
  | N, N, c -> if c = 0 then N
              else C(c mod p, addi (N, N, c/p))
  | C(h, t), N, c -> addi (C(h, t), C(0, N), c)
  | N, C(h, t), c -> addi (C(0, N), C(h, t), c)
  | C(hi, ti), C(hj, tj), c ->
      let res = hi + hj + c in
      C(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
  | N, d, c -> if c = 0 then N
              else C(c mod p, mult1 (N, d, c/p))
  | C(hi, ti), d, c ->
      let res = hi * d + c in
      C(res mod p, mult1 (ti, d, res / p))

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

5.3 Equality Revisited

Now that we have recursive functions on coinductive types, we might ask whether it would be easier to program equality as defined in §3. The answer is yes.

The function is built in much the same way as the `equali` function, with the `appears(false)` solver and an auxiliary function. This is a general trend for coinductive equality: two elements are equal unless there is evidence that they are unequal. This definition corresponds perfectly to the use of the `appears(false)` solver.

The code can be found below, with the small simplification of having expressions on pairs instead of general tuples.

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

let corec[appears(false)] equal_aux arg = match arg with
  | (Var x1, env1), (Var x2, env2) ->
      equal_aux ((assoc x1 env1, env1), (assoc x2 env2, env2))
  | (Var x1, env1), s2 -> equal_aux ((assoc x1 env1, env1), s2)
  | s1, (Var x2, env2) -> equal_aux (s1, (assoc x2 env2, env2))
  | (Int i1, env1), (Int i2, env2) -> i1 = i2
  | (Inj (inj1, e3), env1), (Inj (inj2, e4), env2) ->
      inj1 = inj2 && equal_aux ((e3, env1), (e4, env2))
  | (Pair(e1, e2), env1), (Pair(e3, e4), env2) ->
      equal_aux ((e1, env1), (e3, env1)) &&
      equal_aux ((e1, env1), (e3, env1))
  | _ -> failwith "type error"

let equal arg = not (not_equal arg)
```

5.4 Other Examples

Besides the examples presented here, we have ported the examples presented in [11] 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.

To complement the example of in §1 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)
```

where `fv` is the free variable function defined in the introduction. (For simplicity, we have omitted the case of function abstraction, since it is not relevant for the example.)

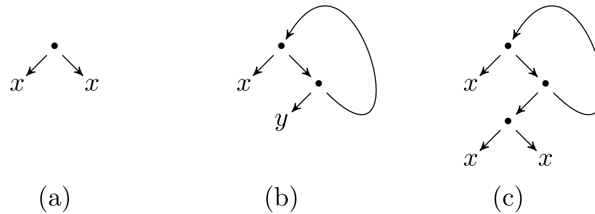
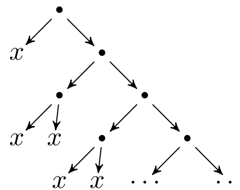


Figure 1: A substitution example.

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 the desired function:

```
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 Limitations and Future Work

In the current implementation, solvers are implemented directly in the interpreter. As versatile as these solvers are, we would like to provide means for programmers to define their own solvers. For that we need to provide tools to manipulate the abstract syntax tree. This is necessary because in general, the right-hand side of an equation is only partially evaluated, thus can be an expression that is not a value. Along with giving this power to the user, we must also provide 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.

As of this writing, the interpreter only allows recursive functions on coalgebraic datatypes that take one argument. Since this argument can be a tuple, this is not much of a limitation, but it would still be nice to remove this restriction.

Typing recursive functions on coalgebraic objects as if they were normal recursive functions seems natural. However, the example of §5.1.6 suggests that the type of the solver may need to be taken into account. However, a solver that transforms abstract syntax trees is not a CoCaml function in the normal sense. We would like to study what it means for it to be well-formed and have a type, and how this type should be used.

Finally, we would like to develop methods for proving the correctness of the implementation of recursive functions on coalgebraic data.

7 Related Work

Syme [14] 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," a gap that we have tried to fill in [11] and here.

Sperber and Thiemann [13] 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.

Hirschowitz, Leroy, and Wells [9] 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.

The closest to our work is the recent paper by Widemann [17], 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.

The question of equality of circular data structures in OCaml has been subject of investigation in, e.g, [8], 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.

8 Conclusions

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. We have shown though numerous examples that 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.

Acknowledgments

Thanks to Bob Constable, Nate Foster, Edgar Friendly, Helle Hvid Hansen, Bart Jacobs, Jonathan Kimmitt, Andrew Myers, Stefan Milius, Ross Tate, and Baltasar Trancón y Widemann for stimulating discussions.

References

[1] Jiří Adámek, Dominik Lücke, and Stefan Milius. Recursive coalgebras of finitary functors. *Theoretical Informatics and Applications*, 41:447–462, 2007.

[2] Jiří Adámek, Stefan Milius, and Jiří Velebil. Elgot algebras. *Log. Methods Comput. Sci.*, 2(5:4):1–31, 2006.

[3] Andrew Baker. 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.

[4] Venanzio Capretta, Tarmo Uustalu, and Varmo Vene. Corecursive algebras: A study of general structured corecursion. In Marcel Vinicius Medeiros Oliveira and Jim Woodcock, editors, *Formal Methods: Foundations and Applications, 12th Brazilian Symp. Formal Methods (SBMF 2009)*, volume 5902 of *Lecture Notes in Computer Science*, pages 84–100, Berlin, 2009. Springer.

[5] Cocaml interpreter. http://www.cs.cornell.edu/Projects/CoCaml/, December 2012.

[6] Adam Eppendahl. Coalgebra-to-algebra morphisms. *Electronic Notes in Theoretical Computer Science*, 29, 1999.

[7] J. A. Goguen and J. W. Thatcher. Initial algebra semantics. In *15th Symp. Switching and Automata Theory*, pages 63–77. IEEE, 1974.

[8] Dmitry Grebeniuk. Library ocaml-cyclist. https://forge.ocamlcore.org/projects/ocaml-cyclist/, June 2010.

[9] Tom Hirschowitz, Xavier Leroy, and J. B. Wells. Compilation of extended recursion in call-by-value functional languages. In *PPDP 2003*, pages 160–171, 2003.

[10] Jean-Baptiste Jeannin and Dexter Kozen. Computing with capsules. In Martin Kutrib, Nelma Moreira, and Rogério Reis, editors, *Proc. Conf. Descriptional Complexity of Formal Systems (DCFS 2012)*, volume 7386 of *Lecture Notes in Computer Science*, pages 1–19, Braga, Portugal, July 2012. Springer.

[11] Jean-Baptiste Jeannin, Dexter Kozen, and Alexandra Silva. Language constructs for non-well-founded computation. Technical Report SEN-1202, CWI Amsterdam, July 2012. 22nd European Symposium on Programming (ESOP'13), to appear.

[12] Dexter Kozen. New. In Ulrich Berger and Michael Mislove, editors, *Proc. 28th Conf. Math. Found. Programming Semantics (MFPS XXVIII)*, pages 13–38, Bath, England, June 2012. Elsevier Electronic Notes in Theoretical Computer Science.

[13] Michael Sperber and Peter Thiemann. ML and the address operator. In *1998 ACM SIGPLAN Workshop on ML*, September 1998.

[14] Don Syme. Initializing mutually referential abstract objects: The value recursion challenge. In *Proc. ACM-SIGPLAN Workshop on ML (2005)*. Elsevier, March 2006.

[15] Paul Taylor. *Practical Foundations of Mathematics*. Number 59 in Cambridge Studies in Advanced Mathematics. Cambridge University Press, 1999.

[16] Wikipedia. p-adic numbers. http://en.wikipedia.org/wiki/P-adic_number, 2012.

[17] Baltasar Trancón y Widemann. Coalgebraic semantics of recursion on circular data structures. In Corina Cirstea, Monika Seisenberger, and Toby Wilkinson, editors, *CALCO Young Researchers Workshop (CALCO-jnr 2011)*, pages 28–42, August 2011.