Programming with Rational Coinductive Streams

Jean-Baptiste Jeannin

University of Michigan

Abstract

Some ML-family languages such as OCaml allow the definition of coinductive types as well as their inhabitants using a `let rec` construct. An example of such elements is coinductive streams. However, there is very little that one can do to manipulate them: pattern matching on the top elements is possible, but recursion will never finish. More importantly, it is virtually impossible to generate coinductive elements on the fly. Recent work with the CoCaml language has started fixing this issue, but it is limited to *regular* coinductive types, that is where the elements eventually point to themselves. For coinductive streams, this corresponds to streams that have a pattern that eventually repeats (e.g., `[3;4;5;1;2;1;2;1;2]`. In this talk we will show initial work on how to build on well-established theory on *rational* streams – a much more comprehensive family of streams – to extend CoCaml to deal with those rational streams.

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 and hence all the theoretical developments have not yet fully found their place in mainstream implementations.

Despite infinite objects being hard to tackle in general, several classes of them are more manageable. For instance, ultimately periodic infinite lists or infinite trees with finitely many subtrees should offer fewer challenges than lists/trees with no apparent regular structure. This class of coinductively defined objects has been widely studied in the literature under the name of regular/rational types. This class was studied before in work on the CoCaml language. More general classes can also be managed, such as rational coinductive streams.

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

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

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

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

Although the `let rec` construct allows us to specify regular infinite lists, further investigation reveals a major shortcoming. For example, suppose we wanted to define a function that, given an infinite list, returns the set of its

elements. For the lists ones and alt, the function should return the sets $\{1\}$ and $\{1, 2\}$, respectively. Note that by regularity, the set of elements is always finite. One would like to write a function definition using equations that pattern-match on the two constructors of the list datatype:

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

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

2 CoCaml: Coinductive Programming with Regular Coinductive Types

Previous work has developed CoCaml [JKS13, JKS17], an extension of OCaml in which functions defined by recursive equations can be supplied with an extra parameter, namely a solver for the equations generated when the function is applied to an argument which might be a regular coinductive object. The contributions of CoCaml can be summarized as follows:

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

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

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

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

Note that the definition of set is not corecursive, as we are not asking for a greatest solution or a unique solution in a final coalgebra, but rather a least solution in a different ordered domain from the one provided by the standard semantics of recursive functions. The standard semantics of recursive functions gives us the least solution in the flat Scott domain with bottom element $\perp$ representing nontermination, whereas we would like the least solution in a different CPO, namely $(\mathcal{P}(\mathbb{Z}), \subseteq)$ with bottom element $\emptyset$.

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

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

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

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

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

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

The construct corec with the parameter iterator [] specifies to the compiler that the equations above should be solved using an iterator—in this case a least fixpoint computation—starting with the initial element []. The CoCaml extended compiler will generate two equations (nodes are given fresh names),

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

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

3 Rational Streams

However, an important drawback of CoCaml is that it is limited to *regular* coinductive types, rendering its uses limited. In this Section we develop initial ideas on how to leverage previous work to extend CoCaml to *rational* coinductive streams. (We will look at more general rational coinductive types in future work.)

A well-established theory of rational streams [Rut01, Rut02, Rut03, Rut08] shows that a much larger class of streams can be represented in a finite way. In particular, in [Rut03], Jan Rutten shows two different ways of defining and representing *rational* streams:

- as sets of stream differential equations; or
- as weighted stream automata.

He also shows that both representations are equivalent, in the sense that the possibilities of both representations are exactly the set of rational streams.

For example, the stream of Fibonacci numbers can be represented as a set of stream differential equations [Rut03]:

$$\sigma'' = \sigma' + \sigma \qquad \sigma(0) = 0 \qquad \sigma'(0) = 1.$$

What is crucial here is that rational coinductive streams have a *finite representation* in a data structure, allowing them to be represented and manipulated by an interpreter or a compiler. In this talk we will show initial work on how to take advantage of this finite representation to extend CoCaml to rational coinductive streams.

References

[JKS13] Jean-Baptiste Jeannin, Dexter Kozen, and Alexandra Silva. Language constructs for non-well-founded computation. In Matthias Felleisen and Philippa Gardner, editors, *22nd European Symposium on Programming (ESOP 2013)*, volume 7792 of *Lecture Notes in Computer Science*, pages 61–80, Rome, Italy, March 2013. Springer.

[JKS17] Jean-Baptiste Jeannin, Dexter Kozen, and Alexandra Silva. Cocaml: Functional programming with regular coinductive types. *Fundamenta Informaticae*, 150(3-4):347–377, 2017.

[Rut01] Jan JMM Rutten. Elements of stream calculus:(an extensive exercise in coinduction). *Electronic Notes in Theoretical Computer Science*, 45:358–423, 2001.

[Rut02] Jan JMM Rutten. Coinductive counting: bisimulation in enumerative combinatorics. *Electronic Notes in Theoretical Computer Science*, 65(1):286–304, 2002.

[Rut03] Jan JMM Rutten. Behavioural differential equations: a coinductive calculus of streams, automata, and power series. *Theoretical Computer Science*, 308(1-3):1–53, 2003.

[Rut08] Jan JMM Rutten. Rational streams coalgebraically. *arXiv preprint arXiv:0807.4073*, 2008.