

# Language Constructs for Non-Well-Founded Computation

Jean-Baptiste Jeannin    Dexter Kozen  
 Cornell University  
 jeannin@cs.cornell.edu    kozen@cs.cornell.edu

Alexandra Silva  
 Radboud Universiteit Nijmegen and CWI Amsterdam  
 alexandra@cs.ru.nl

## Abstract

Recursive functions defined on a coalgebraic datatype  $C$  may not converge if there are cycles in the input, that is, if the input object is not well-founded. Even so, there is often a useful solution; for example, the free variables of an infinitary  $\lambda$ -term, or the expected running time of a finite-state probabilistic protocol.

Theoretical models of recursion schemes have been well studied under the names well-founded coalgebras, recursive coalgebras [2], corecursive algebras [4], and Elgot algebras [1]. Much of this work focuses on conditions ensuring unique or canonical solutions, e.g. when  $C$  is well-founded.

If  $C$  is not well-founded, then there can be multiple solutions. The standard semantics of recursive programs gives a particular solution, namely the least solution in a flat Scott domain, which may not be the one we want. Unfortunately, current programming languages provide no support for specifying alternatives.

In this paper we give numerous examples in which it would be useful to do so: free variables,  $\alpha$ -conversion, and substitution in infinitary terms; halting probabilities, expected running times, and outcome functions of probabilistic protocols; various functions on streams; semantics of alternating automata and games; and abstract interpretation. In each case the function would diverge under the standard semantics of recursion. We propose programming language constructs that would allow the specification of alternative solutions and methods to compute them. The programmer must essentially provide a way to solve equations in the codomain. We show how to implement these new constructs as an extension of OCaml and give implementations of all our examples. In some cases, some of the work can be automated.

We also prove some theoretical results characterizing well-founded coalgebras that slightly extend results of Adamek, Luecke, and Milius [2].

**Categories and Subject Descriptors** F.3.2 [Theory of Computation]: Semantics of Programming Languages; F.3.3 [Theory of Computation]: Studies of Program Constructs

**General Terms** Program constructs, Coinduction

**Keywords** coalgebra, coinduction, coinductive datatypes, recursion, programming language semantics

## 1. Introduction

Coalgebraic datatypes have become popular in recent years in the study of infinite behaviors and non-terminating computation. One would like to define functions on coinductive datatypes by structural recursion, but such functions may not converge if there are cycles in the input; that is, if the input object is not well-founded. Even so, there is often a useful solution that we would like to compute.

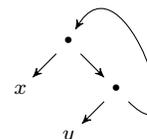
For example, consider the problem of computing the set of free variables of a  $\lambda$ -term. In pseudo-ML, we might write

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

let rec fv = function
  | Var v -> {v}
  | App (t1,t2) -> (fv t1) ∪ (fv t2)
  | Lam (x,t) -> (fv t) - {x}
```

and this works provided the argument is an ordinary (well-founded)  $\lambda$ -term. However, if we call the function on an infinitary term, say

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



(1)

then the function will diverge, even though it is clear the answer should be  $\{x, y\}$ . Note that this is not a corecursive definition: 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 gives us the least solution in the flat Scott domain  $(\mathcal{P}(\mathbf{string})_{\perp}, \sqsubseteq)$  with bottom element  $\perp$  representing nontermination, whereas we would like the least solution in a different CPO, namely  $(\mathcal{P}(\mathbf{string}), \sqsubseteq)$  with bottom element  $\emptyset$ .

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. A few of our examples, like substitution, could be computed by lazy evaluation, but most of them, for example free variables, could not.

Abstractly, the situation is governed by diagrams of the form

$$\begin{array}{ccc}
 C & \xrightarrow{h} & A \\
 \gamma \downarrow & & \uparrow \alpha \\
 FC & \xrightarrow{Fh} & FA
 \end{array}
 \tag{2}$$

where  $(C, \gamma)$  and  $(A, \alpha)$  are a coalgebra and an algebra, respectively, for an endofunctor  $F$ . An  $h$  satisfying this diagram is called

an *F-coalgebra-algebra morphism* [7]. Intuitively, given an input  $x \in C$ , the destructor  $\gamma: C \rightarrow FC$  is applied to  $x$  to compute the arguments to recursive calls (including the identification of base cases), the map  $Fh: FC \rightarrow FA$  performs the recursive calls, and the constructor  $\alpha: FA \rightarrow A$  is applied to the results of the recursive calls to obtain the value of  $h(x)$ .

A canonical example of this abstract definition is the factorial function:

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

The diagram (2) is instantiated to

$$\begin{array}{ccc} \mathbb{N} & \xrightarrow{h} & \mathbb{N} \\ \gamma \downarrow & & \uparrow \alpha \\ \mathbb{1} + \mathbb{N} \times \mathbb{N} & \xrightarrow{\text{id}_{\mathbb{1}} + \text{id}_{\mathbb{N}} \times h} & \mathbb{1} + \mathbb{N} \times \mathbb{N} \end{array} \quad (3)$$

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

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

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  $\gamma$  determines whether the argument is a base case, and if not, prepares the recursive call. The constructor  $\alpha$  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. The free variables example above also fits this scheme with  $FX = \text{Var} + X^2 + \text{Var} \times X$  and

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

Theoretical accounts of this general idea have been well studied under the names well-founded coalgebras [13], recursive coalgebras [2], corecursive algebras [4], and Elgot algebras [1]. Also relevant from a more general perspective are iteration theories [3] and traced monoidal categories [9]. Most of this work is focused on conditions ensuring unique solutions, primarily when  $C$  is well-founded or when  $A$  is a final coalgebra. The account most relevant to this study is [1], in which a canonical solution can be specified even when it is not unique, provided various desirable conditions are met; for example, when  $A$  is a complete CPO and  $\alpha$  is continuous, or when  $A$  is a complete metric space and  $\alpha$  is contractive. We would like to bring this theoretical work to bear on programming language design and implementation.

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: it is the least solution in the standard flat Scott domain. For example, the factorial function is uniquely defined by (3) in this sense. If  $C$  is not well-founded, there can be multiple solutions, and the standard one may not be the one we want. Nevertheless, the diagram (2) can still serve as a valid definitional scheme, provided we are allowed to specify a desired solution. In the free variables example, the codomain of the function (sets of variables) is indeed a complete CPO under the usual set inclusion order, and the constructor  $\alpha$  is continuous, thus the desired solution can be obtained by a least fixpoint computation.

Unfortunately, current programming languages provide little support for specifying alternative solutions. One must be able to specify a canonical method for solving systems of equations over an  $F$ -algebra (the codomain) obtained from the function definition and the input. We will demonstrate through many examples that

such a feature would be extremely useful in a programming language and would bring coinduction and coinductive datatypes to a new level of usability in accordance with the elegance already present for algebraic datatypes. Our examples include free variables,  $\alpha$ -conversion, and substitution in infinitary terms; halting probabilities, expected running times, and outcome functions of probabilistic protocols; various functions on streams; semantics of alternating automata and games; and abstract interpretation. In each case, the function would diverge under the standard semantics of recursion. We propose programming language constructs that would allow the specification of alternative solutions and methods to compute them. These examples require different solution methods: iterative least fixpoint computation, Gaussian elimination, structural coinduction. We describe how this feature might be implemented in a functional language and give mock-up implementations of our examples. In our implementation, we show how to automatically extract a system of equations from the function definition and the input, which can then be passed to a standard solver. In many cases the process can be largely automated, requiring little extra work on the part of the programmer.

An orthogonal issue is that current functional languages are not particularly well suited to the manipulation of coinductive datatypes. For example, in OCaml one can form coinductive objects with **let rec** as in (1), but due to the absence of mutable variables such objects can only be created and not dynamically manipulated, which severely limits their usefulness. One can simulate them with references, but this negates the elegance of algebraic manipulation of inductively defined datatypes, for which the ML family of languages is so well known. It would be of benefit to be able to treat coinductive types the same way.

We also include some theoretical results that clarify and slightly generalize some results of [2], namely:

- Every  $F$ -coalgebra  $C$  contains a maximal well-founded sub-coalgebra  $\text{wf } C$ .
- If  $R$  is a final  $F$ -coalgebra, then  $\text{wf } R$  is the initial  $F$ -algebra.
- Let  $C$  be an  $F$ -coalgebra. The following are equivalent:
  - (i)  $C$  is well-founded; that is,  $C = \text{wf } C$ .
  - (ii) The induction principle
$$\frac{\square P \rightarrow P}{P}$$
is valid for  $C$ .
  - (iii) There is a unique coalgebra morphism  $C \rightarrow \text{wf } R$ .
  - (iv) There is a unique coalgebra-algebra morphism from  $C$  to any  $F$ -algebra.
- Let  $(A, \alpha)$  be an ordered  $F$ -algebra such that  $A$  is a chain-complete and  $\alpha$  order-continuous. The construction of the least solution  $h$  is natural in  $S$ ; that is, if  $f: S \rightarrow S'$  is an  $F$ -coalgebra morphism, then  $h_S = h_{S'} \circ f$ .

The last item provides a method to automatically compute least solutions in an ordered setting and is therefore of practical importance, as we will show later in §A.

Our full implementation with all examples and solvers is available from <http://www.cs.cornell.edu/~kozen/nwf/jb.zip>. A more elementary preliminary version with only the free variable and probabilistic examples and iteration solver is available from <http://www.cs.cornell.edu/~kozen/nwf/dk.zip>.

## 2. Motivating Examples

In this section we present a number of motivating examples that illustrate the diversity of the problem we are trying to tackle. The ex-

amples can be divided into two subclasses: well-founded and non-well-founded. The latter is no doubt the more challenging, but the former illustrates in a simple setting the general idea of coalgebra-algebra homomorphisms, so we start with such an example: computing the greatest common divisor of two numbers. Further examples of well-founded definitions can be found in the cited literature, including the Fibonacci function and various divide-and-conquer algorithms such as quicksort and mergesort. Yet another interesting example involving mutual recursive functions can be handled by generalizing the results of [2] to a multi-sorted setting. We will discuss this in appendix A. We illustrate it with mutually recursive definitions of the even/odd predicates on natural numbers in section 2.6, and the more intricate Ackermann function in section 2.8.

## 2.1 Integer GCD

For integers  $m, n \geq 0$  but not both 0, we would like to compute a triple  $(g, s, t)$  such that  $g$  is the greatest common divisor (gcd) of  $m$  and  $n$  and  $sm + tn = g$ . A recursive definition is

```
let rec gcd m n =
  if n = 0 then (m,1,0) else
  let (q,r) = (m/n, m mod n) in
  let (g,s,t) = gcd n r in
  (g,t,s-q)
```

This gives the following instantiation of (2):

$$\begin{array}{ccc} \mathbb{N} \times \mathbb{N} & \xrightarrow{h} & \mathbb{N} \times \mathbb{Z} \times \mathbb{Z} \\ \gamma \downarrow & & \uparrow \alpha \\ F(\mathbb{N} \times \mathbb{N}) & \xrightarrow{Fh} & F(\mathbb{N} \times \mathbb{Z} \times \mathbb{Z}) \end{array}$$

Here  $FX = \mathbb{N} + X \times \mathbb{N}$  and

$$\gamma(m, n) = \begin{cases} \iota_0(m) & \text{if } n = 0 \\ \iota_1(n, m \bmod n, m/n) & \text{if } n \neq 0 \end{cases}$$

$$\alpha(\iota_0(g)) = (g, 1, 0)$$

$$\alpha(\iota_1(g, s, t, q)) = (g, t, s - q).$$

The theory of recursive coalgebras [2] guarantees the existence of a unique function satisfying the diagram.

## 2.2 Substitution

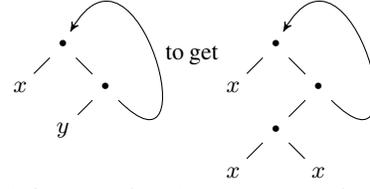
We now move to non-well-founded examples, with another function on infinitary  $\lambda$ -terms: substitution of a variable  $y$  by a term  $t$ . A typical implementation would be

```
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)
| Lam (x,s) ->
  if x = y then Lam (x,s)
  else if x ∈ fv t then
    let w = fresh ()
    in Lam (w, subst t y (rename w x s))
  else Lam (x, subst t y s)
```

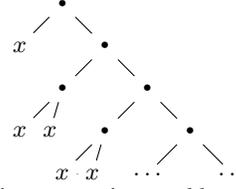
where  $fv$  is the free variable function defined above and  $rename\ w\ x\ s$  is a function that renames a variable  $x$  into  $w$  in a term  $s$ .

```
let rec rename w y = function
| Var x -> Var (if x = y then w else x)
| App (t1,t2) -> App (rename w y t1, rename w y t2)
| Lam (x,s) ->
  if x = y then Lam (x,s)
  else Lam (x, rename w y s)
```

An example of application would be to replace  $y$  by  $\begin{array}{c} \bullet \\ / \quad \backslash \\ x \quad x \end{array}$  in



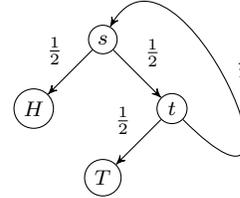
The usual semantics would



This computation would never finish.

## 2.3 Probabilistic Protocols

In this subsection, we present a few examples in the realm of probabilistic protocols. 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.

$$\Pr_H(s) = \frac{1}{2} + \frac{1}{8} + \frac{1}{32} + \frac{1}{128} + \dots = \frac{2}{3}$$

$$\Pr_H(t) = \frac{1}{4} + \frac{1}{16} + \frac{1}{64} + \frac{1}{256} + \dots = \frac{1}{3}.$$

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

$$\Pr_H(s) = \frac{1}{2} + \frac{1}{2} \cdot \Pr_H(t) \quad \Pr_H(t) = \frac{1}{2} \cdot \Pr_H(s).$$

This gives rise to a contractive map on the unit interval, which has a unique solution. It is also monotone and continuous with respect to the natural order on the unit interval. In either case an iterative solution is possible, provided equality on the codomain is defined to within  $\varepsilon$ . One would like to write a recursive program, say something like

```
let rec pr_heads s = function
| H -> 1.
| T -> 0.
| Flip (p,u,v) ->
  p *. (pr_heads u) +. (1 -. p) *. (pr_heads v)
```

and specify that the extracted equations should be solved by Gaussian elimination or iteration. We give implementations using both methods.

The protocol described above is an instance of a class of probabilistic protocols that can be described by coalgebras  $(S, \beta)$ , where  $\beta: S \rightarrow \Gamma + S^\Sigma$ , and  $\Sigma$  and  $\Gamma$  are the input and output alphabets, respectively. The *von Neumann trick* for simulating a fair coin with a coin of arbitrary bias is another example of such a protocol. The same approach could be used to model other probabilistic processes such as Markov chains and Markov decision processes.

Next, we discuss two other examples of interesting functions in the probabilistic realm.

The expected consumption of random bits starting from state  $s$  is the least solution of the equation

$$E(s) = \begin{cases} 0 & \text{if } \beta(s) \in \{H, T\} \\ 1 + p \cdot E(\beta(s)_0) + (1 - p) \cdot E(\beta(s)_1) & \text{if } \beta(s) \in S^{\{0,1\}}. \end{cases}$$

The *outcome*  $O(s)$  of the simulation starting from state  $s \in S$  of a coalgebra  $(S, \beta)$  as above is a random variable defined on the probability space  $\{0, 1\}^\omega$  taking values in  $\{H, T, \perp\}$ . The value  $\perp$  signifies nonhalting. Formally,

$$O: S \rightarrow \{0, 1\}^\omega \rightarrow \{H, T, \perp\}$$

is the least fixpoint of the equation

$$O(s)(X \cdot \sigma) = \begin{cases} \beta(s) & \text{if } \beta(s) \in \{H, T\} \\ O(\beta(s)_X)(\sigma) & \text{if } \beta(s) \in \{0, 1\} \rightarrow S. \end{cases}$$

Both functions fit the abstract definitional scheme (2), and the solution we are seeking can be described as the least fixpoint in an appropriate ordered domain.

## 2.4 Alternating Turing Machines and IND Programs

The semantics of alternating Turing machines is described in terms of an inductive labeling of machine configurations  $C$  with either  $\mathbf{0}$  (rejecting),  $\mathbf{1}$  (accepting), or  $\perp$  (undetermined). In the present framework, the function  $\gamma$  would give the set of successor configurations and the labeling of the state as either existential or universal, and  $\alpha$  would tell how to label configurations  $\mathbf{0}$ ,  $\mathbf{1}$ , or  $\perp$  inductively up the computation tree. Formally,  $\alpha$  gives the infimum for universal configurations and supremum for existential configurations in 3-valued Kleene logic  $\mathbb{3} = \{\mathbf{0}, \perp, \mathbf{1}\}$  with ordering  $\mathbf{0} \leq \perp \leq \mathbf{1}$ .

$$\begin{array}{ccc} C & \xrightarrow{h} & \mathbb{3} \\ \gamma \downarrow & & \uparrow \alpha \\ 2 \times \mathcal{P}_{\text{fin}}(C) & \xrightarrow{\text{id}_2 + \mathcal{P}_{\text{fin}}(h)} & 2 \times \mathcal{P}_{\text{fin}}(\mathbb{3}) \end{array}$$

The canonical solution is defined to be the least fixpoint with respect to a different order, namely the flat Scott order  $\perp \sqsubseteq \mathbf{0}, \perp \sqsubseteq \mathbf{1}$ . This example is interesting, because it is a case in which  $\alpha$  is not strict; for example, a universal configuration can be labeled  $\mathbf{0}$  as soon as one of its successors is known to be labeled  $\mathbf{0}$ , regardless of the labels of the other successors.

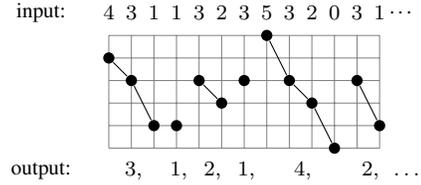
A similar model is the IND programming language for the inductive sets [8]. An IND program consists of a sequence of labeled statements of three kinds: universal and existential assignment ( $x := \forall$  and  $x := \exists$ , respectively), conditional test (if  $s = t$  then  $\ell_1$  else  $\ell_2$ ), and halting (accept, reject). IND programs accept exactly the inductively definable sets, which over  $\mathbb{N}$  are exactly the  $\Pi_1^1$  sets. The semantics is identical to alternating Turing machines, except that the branching degree is equal to the cardinality of the domain of computation, thus the finite powerset functor must be replaced by the unrestricted powerset functor.

## 2.5 Descending Sequences

As the simplest nontrivial coinductive datatype, *streams* offer the ideal playground to test new theories. We present an example on streams of natural numbers  $\mathbb{N}^\omega$ . The following example, taken from a talk by Capretta, has a unique solution, but does not fit the existing theory of well-founded coalgebras.

The goal is to produce from a given stream of natural numbers another stream of natural numbers containing the lengths of the maximal strictly descending subsequences of the input stream. An

example is shown in the following figure, where the input stream is depicted in a grid to easily picture the order of elements.



Here is a simple recursive definition of the function:

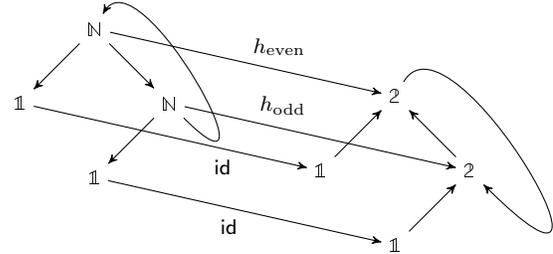
```
let downruns s =
  let rec countdown n (x::y::r) =
    if x > y then countdown (n+1) (y::r)
    else n :: (countdown 1 (y::r)) in
  countdown 1 s
```

## 2.6 Mutually Recursive Functions: even-odd

This subsection illustrates how to generalize the uniqueness result of [2] to the multi-sorted case and how this generalization can be used to handle mutually recursive functions. We show the simplest example: the definition of the even and odd predicates on natural numbers.

```
let rec even n = if n = 0 then true else odd (n-1)
and odd n = if n = 0 then false else even (n-1)
```

We can depict the recursion graphically with the following diagram:



A way to look at this is as an endofunctor  $F: \text{Set}^V \rightarrow \text{Set}^V$ , where  $V = \{\text{even}, \text{odd}\}$ . The functor is defined by:  $F(A, B) = (\mathbb{1} + B, \mathbb{1} + A)$  and if  $g: A \rightarrow A'$  and  $h: B \rightarrow B'$ , then  $F(g, h) = (\text{id} + h, \text{id} + g): F(A, B) \rightarrow F(A', B')$ .

An  $F$ -coalgebra is a pair  $((C, D), \gamma)$ , where  $\gamma: (C, D) \rightarrow F(C, D)$  is a morphism in the underlying category  $\text{Set}^V$ ; that is,

$$\gamma = (\gamma_{\text{even}}, \gamma_{\text{odd}}): (C, D) \rightarrow (\mathbb{1} + D, \mathbb{1} + C),$$

where  $\gamma_{\text{even}}: C \rightarrow \mathbb{1} + D$  and  $\gamma_{\text{odd}}: D \rightarrow \mathbb{1} + C$ . Similarly, an  $F$ -algebra is a pair  $((A, B), \alpha)$ , where  $\alpha: F(A, B) \rightarrow (A, B)$  is a morphism in  $\text{Set}^V$ ; that is,

$$\alpha = (\alpha_{\text{even}}, \alpha_{\text{odd}}): (\mathbb{1} + B, \mathbb{1} + A) \rightarrow (A, B),$$

where  $\alpha_{\text{even}}: \mathbb{1} + B \rightarrow A$  and  $\alpha_{\text{odd}}: \mathbb{1} + A \rightarrow B$ .

An  $F$ -algebra-coalgebra morphism  $h: ((C, D), \gamma) \rightarrow ((A, B), \alpha)$  is a map  $h = (h_{\text{even}}, h_{\text{odd}}): (C, D) \rightarrow (A, B)$  such that the following diagram commutes:

$$\begin{array}{ccc} (C, D) & \xrightarrow{(h_{\text{even}}, h_{\text{odd}})} & (A, B) \\ (\gamma_{\text{even}}, \gamma_{\text{odd}}) \downarrow & & \uparrow (\alpha_{\text{even}}, \alpha_{\text{odd}}) \\ (\mathbb{1} + D, \mathbb{1} + C) & \xrightarrow{(\text{id} + h_{\text{odd}}, \text{id} + h_{\text{even}})} & (\mathbb{1} + B, \mathbb{1} + A) \end{array}$$

In our application, we have  $A = B = 2$  and  $C = D = \mathbb{N}$ , with

$$\gamma_{\text{even}}(n) = \gamma_{\text{odd}}(n) = \begin{cases} \iota_0() & \text{if } n = 0 \\ \iota_1(n-1) & \text{if } n > 0 \end{cases}$$

$$\alpha_{\text{even}}(\iota_0()) = \mathbf{1} \quad \alpha_{\text{odd}}(\iota_0()) = \mathbf{0} \quad \alpha_{\text{even}}(\iota_1(b)) = \alpha_{\text{odd}}(\iota_1(b)) = b.$$

## 2.7 Abstract Interpretation

This section presents an example involving abstract interpretation of a simple imperative language, following Cousot and Cousot [6] and further inspired by lecture notes of Stephen Chong [5].

Consider a simple imperative language of while programs with integer expressions  $a$  and commands  $c$ . Let  $\text{Var}$  be a countable set of variables.

$$a ::= n \in \mathbb{Z} \mid x \in \text{Var} \mid a_1 + a_2$$

$$c ::= \text{skip} \mid x := a \mid c_1 ; c_2 \mid \text{if } a \text{ then } c_1 \text{ else } c_2 \mid \text{while } a \text{ do } c$$

For the purpose of tests in the conditional and while loop, an integer is considered true if and only if it is nonzero. Otherwise, the operational semantics is standard, in the style of [14]. A store is a partial function from variables to integers, an arithmetic expression is interpreted relative to a store and returns an integer; and a command is interpreted relative to a store and returns an updated store.

Abstract interpretation defines an abstract domain that approximates the values manipulated by the program. We define an abstract domain for integers that abstracts an integer by its sign. The set of abstract values is  $\text{AbsInt} = \{\text{neg}, \text{zero}, \text{pos}, \top\}$ , where  $\text{neg}$ ,  $\text{zero}$ , and  $\text{pos}$  represent negative, zero, and positive integers, respectively, and  $\top$  represents an integer of unknown sign. The abstract values form a join semilattice with join  $\sqcup$ :

$$\begin{array}{c} \top \\ \swarrow \quad \downarrow \quad \searrow \\ \text{neg} \quad \text{zero} \quad \text{pos} \end{array} \quad (4)$$

The abstract interpretation of an arithmetic expression is defined relative to an abstract store  $\sigma : \text{Var} \rightarrow \text{AbsInt}$ , used to interpret the abstract values of variables. We write  $\text{AS} = \text{Var} \rightarrow \text{AbsInt}$  for the set of abstract stores. The abstract interpretation of arithmetic expressions is given by:

$$\mathcal{A}[[n]]\sigma = \begin{cases} \text{pos} & \text{if } n > 0 \\ \text{zero} & \text{if } n = 0 \\ \text{neg} & \text{if } n < 0 \end{cases}$$

$$\mathcal{A}[[x]]\sigma = \sigma(x)$$

$$\mathcal{A}[[a_1 + a_2]]\sigma = \mathcal{A}[[a_1]]\sigma \sqcup \mathcal{A}[[a_2]]\sigma.$$

The abstract interpretation of commands returns an abstract store, which is an abstraction of the concrete store returned by the commands. Abstract stores form a join semilattice, where the join  $\sqcup$  of two abstract stores just takes the join of each variable:

$$(\sigma_1 \sqcup \sigma_2)(x) = \sigma_1(x) \sqcup \sigma_2(x).$$

Commands other than the while loop are interpreted as follows:

$$\mathcal{C}[[\text{skip}]]\sigma = \sigma$$

$$\mathcal{C}[[x := a]]\sigma = \sigma[x \mapsto \mathcal{A}[[a]]\sigma]$$

$$\mathcal{C}[[c_1 ; c_2]]\sigma = \mathcal{C}[[c_2]](\mathcal{C}[[c_1]]\sigma)$$

$$\mathcal{C}[[\text{if } a \text{ then } c_1 \text{ else } c_2]]\sigma = \begin{cases} \mathcal{C}[[c_1]]\sigma & \text{if } \mathcal{A}[[a]]\sigma \in \{\text{pos}, \text{neg}\} \\ \mathcal{C}[[c_2]]\sigma & \text{if } \mathcal{A}[[a]]\sigma = \text{zero} \\ \mathcal{C}[[c_1]]\sigma \sqcup \mathcal{C}[[c_2]]\sigma & \text{otherwise} \end{cases}$$

We would ideally like to define the abstract interpretation of the while loop as:

$$\mathcal{C}[[\text{while } a \text{ do } c]]\sigma = \begin{cases} \sigma & \text{if } \mathcal{A}[[a]]\sigma = \text{zero} \\ \sigma \sqcup \mathcal{C}[[\text{while } a \text{ do } c]](\mathcal{C}[[c]]\sigma) & \text{otherwise} \end{cases}$$

Unfortunately, in the case where  $\mathcal{A}[[a]]\sigma \neq \text{zero}$ , this is not a well-founded definition of  $\mathcal{C}[[\text{while } a \text{ do } c]]$ , because it is possible for  $\sigma$  and  $\mathcal{C}[[c]]\sigma$  to be equal. However, it is a correct definition of  $\mathcal{C}[[\text{while } a \text{ do } c]]$  as a least fixpoint in the join semilattice of abstract stores. The existence of the least fixpoint can be obtained in a finite time by iteration because the join semilattice of abstract stores satisfies the ascending chain condition (ACC), that is, it does not contain any infinite ascending chains.

Given  $\mathcal{A}[[a]]$  and  $\mathcal{C}[[c]]$  previously defined,  $\mathcal{C}[[\text{while } a \text{ do } c]]$  can be defined by the following diagram:

$$\begin{array}{ccc} \text{AS} & \xrightarrow{\mathcal{C}[[\text{while } a \text{ do } c]]} & \text{AS} \\ \gamma \downarrow & & \uparrow \alpha \\ \text{AS} + \text{AS} \times \text{AS} & \xrightarrow{\text{id}_{\text{AS}} + \text{id}_{\text{AS}} \times \mathcal{C}[[\text{while } a \text{ do } c]]} & \text{AS} + \text{AS} \times \text{AS} \end{array}$$

where the functor is  $FX = \text{AS} + \text{AS} \times X$  and

$$\gamma(\sigma) = \begin{cases} \iota_1(\sigma) & \text{if } \mathcal{A}[[a]]\sigma = \text{zero} \\ \iota_2(\sigma, \mathcal{C}[[c]]\sigma) & \text{otherwise} \end{cases}$$

$$\alpha(\iota_1(\sigma)) = \sigma$$

$$\alpha(\iota_2(\sigma, \tau)) = \sigma \sqcup \tau$$

The function  $\mathcal{C}[[\text{while } a \text{ do } c]]$  is the least function in the pointwise order that makes the above diagram commute.

This technique allows us to define  $\mathcal{C}[[c]]$  in general, inductively on the structure of  $c$ . An inductive definition can be used here because the set of abstract syntax trees is well-founded.

The literature on abstract interpretation explains how to compute the least fixpoint, and much research has been done on techniques for accelerating convergence to the least fixpoint. This body of research can inform compiler optimization techniques for computation with coalgebraic types.

## 2.8 Ackermann's Function

The Ackermann function

$$A(0, n) = n + 1$$

$$A(m + 1, 0) = A(m, 1) \quad (5)$$

$$A(m + 1, n + 1) = A(m, A(m + 1, n))$$

is a notoriously fast-growing function that also fits into our general scheme (although one should not try to compute it!). This example is quite interesting, because at first glance it seems not to fit into the general scheme (2) because of the nested recursive call in the third clause. However, a key insight comes from the termination proof, which is done by induction on the well-founded lexicographic order on  $\mathbb{N} \times \mathbb{N}$  with  $m$  as the more significant parameter. We see that we can break the definition into two stages, both higher-order.

Rewriting  $A(m, n)$  as  $A_m(n)$ , we have that (5) is equivalent to

$$A_0 = \lambda n. n + 1 \quad A_{m+1} = \lambda n. A_m^{n+1}(1),$$

where  $f^n$  denotes the  $n$ -fold composition of  $f$  with itself:

$$f^0 = \lambda n. n \quad f^{n+1} = f \circ f^n.$$

The outermost stage computes  $m \mapsto A_m$ . The diagram is

$$\begin{array}{ccc} \mathbb{N} & \xrightarrow{A} & \mathbb{N}^{\mathbb{N}} \\ \gamma \downarrow & & \uparrow \alpha \\ \mathbb{1} + \mathbb{N} & \xrightarrow{\text{id}_{\mathbb{1}} + A} & \mathbb{1} + \mathbb{N}^{\mathbb{N}} \end{array}$$

where

$$\begin{aligned} \gamma(0) &= \iota_0() & \alpha(\iota_0()) &= \lambda n. n + 1 \\ \gamma(m + 1) &= \iota_1(m) & \alpha(\iota_1(f)) &= \lambda n. f^{n+1}(1). \end{aligned}$$

In turn, the function  $\alpha$  is defined in terms the  $n$ -fold composition function  $(n, f) \mapsto f^n$ :

$$\begin{array}{ccc} \mathbb{N} \times D^D & \xrightarrow{\text{comp}} & D^D \\ \gamma \downarrow & & \uparrow \alpha \\ F(\mathbb{N} \times D^D) & \xrightarrow{F(\text{comp})} & F(D^D) \end{array}$$

where  $F X = \mathbb{1} + D^D \times X$  and

$$\begin{aligned} \gamma(0, f) &= \iota_0() & \alpha(\iota_0()) &= \text{id}_D \\ \gamma(n + 1, f) &= \iota_1(f, n, f) & \alpha(\iota_1(f, g)) &= f \circ g. \end{aligned}$$

### 3. A Framework for Non-Well-Founded Computation

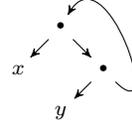
In this section we discuss our proposed framework for incorporating language constructs to support non-well-founded computation. At a high level, we wish to specify a function  $h$  uniquely using a finite set  $E$  of structural recursive equations. The function is defined in much the same way as an ordinary recursive function on an inductive datatype. However, the value  $h(x)$  of the function in a particular input  $x$  is computed not by calling the function, but by generating a system of equations from the function definition and then passing the equations to a specified equation solver to find a solution. The equation solver is either a standard library function or programmed by the user according to an explicit interface.

The process is partitioned into several tasks as follows.

1. The left-hand sides of the clauses in the function definition determine syntactic terms representing equation schemes. These schemes are extracted by the compiler from the abstract syntax tree of the left-hand side expressions. This determines (more or less, subject to optimizations) the function  $\gamma$  in the diagram (2).
2. The right-hand sides of the clauses in the function definition determine the function  $\alpha$  in the diagram (2) (again, more or less, subject to optimizations). These expressions essentially tell how to evaluate terms extracted in step 1 in the codomain. As in 1, these are determined by the compiler from the abstract syntax trees of the right-hand sides.
3. At runtime, when the function is called with a coalgebraic element  $c$ , a finite system of equations is generated from the schemes extracted in steps 1 and 2, one equation for each element of the coalgebra reachable from  $c$ . In fact, we can take the elements reachable from  $c$  as the variables in our equations. Each such element matches exactly one clause of the function body, and this determines the right-hand side of the equation that is generated.
4. The equations are passed to a solver that is specified by the user. This will presumably be a module that is programmed separately according to a fixed interface and available as a library function. There should be a simple syntactic mechanism

for specifying an alternative solution method (although we do not specify here what that should look like).

Let us illustrate this using our initial example of the free variables. Recall the infinitary  $\lambda$ -term below and the definition of the free variables function from the introduction:



```

let rec fv = function
| Var v -> {v}
| App (t1,t2) -> (fv t1) ∪ (fv t2)
| Lam (x,t) -> (fv t) - {x}

```

(6)

Steps 1 and 2 would analyze the left-and right-hand sides of the three clauses in the body at compile time to determine the equation schemes. Then at runtime, if the function were called on the coalgebraic element pictured, the runtime system would generate four equations, one for each node reachable from the top node:

$$\begin{aligned} \text{fv } t &= (\text{fv } x) \cup (\text{fv } u) \\ \text{fv } u &= (\text{fv } y) \cup (\text{fv } t) \\ \text{fv } x &= \{x\} \\ \text{fv } y &= \{y\} \end{aligned}$$

where  $t$  and  $u$  are the unlabeled top and right nodes of the term above.

As noted, these equations have many solutions. In fact, any set containing the variables  $x$  and  $y$  will be a solution. However, we are interested in the least solution in the ordered domain of sets of variables with bottom element  $\emptyset$ . In this case, the least solution would assign  $x$  to the rightmost node,  $y$  to the lowest node, and  $\{x, y\}$  to the other two nodes.

With this in mind, we would pass the generated equations to an iterative equation solver, which would produce the desired solution. In many cases, such as this example, the codomain is a complete partial order and we have default solvers to compute least fixpoints, leaving to the programmer the simple task of indicating that this is the desired solution method. That would be an ideal situation: the defining equations of (6) plus a simple tag would be enough to obtain the desired solution.

#### 3.1 Generating Equations

The equations are generated from the recursive function definition and the input  $c$ , a coalgebraic element, in accordance with the abstract definitional scheme (2). The variables can be taken to be the elements of the coalgebraic object reachable from  $c$ . There are finitely many of these, as no infinite object can ever exist in a running program. More accurately stated, the objects of the final coalgebra represented by coalgebraic elements during program execution are all *regular* in the sense that they have a finite representation. These elements are first collected into a data structure (in our implementation, simply a list) and the right-hand sides of the equations are determined by the structure of the object using pattern matching. The object matches exactly one of the terms extracted in step 1.

#### 3.2 A Theorem on Well-Foundedness

When the function is called on a well-founded argument, the solution is unique and the standard semantics will terminate. The following theorem characterizes conditions under which this occurs. It slightly generalizes [2] to the non-finitary case.

**THEOREM 3.1.** *Let  $C$  be an  $F$ -coalgebra. The following are equivalent:*

- (i)  $C$  is well-founded; that is,  $C = \text{wf } C$ .

(ii) *The induction principle*

$$\frac{\Box P \rightarrow P}{P}$$

is valid for  $C$ .

(iii) *There is a unique coalgebra morphism  $C \rightarrow \text{wf } R$ .*

(iv) *There is a unique coalgebra-algebra morphism from  $C$  to any  $F$ -algebra.*

The proof of this theorem, which we provide in appendix to not disrupt the main focus of the paper, relies on some extra interesting facts which we also prove, namely the fact that every  $F$ -coalgebra  $C$  contains a maximal well-founded subcoalgebra  $\text{wf } C$  and that if  $R$  is the final  $F$ -coalgebra, then  $\text{wf } R$  is the initial  $F$ -algebra.

However, in many interesting cases,  $h$  is not unique and depends on the choice of solution methods in the codomain  $A$ . For a large class of codomains we have however a canonical solution, namely the least solution. This was studied in [1], although this study was quite abstract and did not consider any programming language constructs or practical applications.

The following theorem essentially says that under certain conditions, the function can actually be considered a function on the final coalgebra and is independent of the representation in the program.

**THEOREM 3.2.** *Let  $(A, \alpha)$  be an ordered  $F$ -algebra such that  $A$  is a chain-complete and  $\alpha$  order-continuous. The construction of the least fixpoint of the map  $h \mapsto \alpha \circ Fh \circ \gamma$ , is natural in  $S$ ; that is, if  $f: S \rightarrow S'$  is an  $F$ -coalgebra morphism, then  $h_S = h_{S'} \circ f$ .*

This covers most of the examples of §2, where the intended solution was the least one. In some of the examples, however, we were interested in other solutions. For instance, in the substitution example we intended to get an element of the final coalgebra; in this case, the solution is unique for other reasons, but there is no natural order on the codomain. Our methodology does not restrict to a particular solution, but instead gives the programmer the power to choose.

## 4. Implementation

The examples of the previous section show the need for new program constructs that would allow the user to manipulate corecursive types with the same ease and elegance as we are used to for algebraic datatypes. It is the goal of this section to provide language constructs that allow us to provide the intended semantics to the examples above in a functional language like OCaml.

The general idea behind the implementation is as follows. We want to keep the overhead for the programmer at a minimum level. We want the programmer to specify the function in the usual way, then at runtime, when the function is evaluated on a given argument, a set of equations is generated and passed on to a solver, which will find a solution according to the specification. In an ideal situation, the programmer only has to specify the solver. For the examples where a CPO structure is present in the codomain, such as the free variable case, or when we have a complete metric space, we provide the typical solution methods (least and unique fixpoint) and the programmer only needs to tag the codomain with the intended solver. In other cases, the programmer needs to implement the solver.

### 4.1 Equations and Solvers

Recall the general diagram governing the situation:

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

Our implementation aims at allowing the programmer to encode this diagram as an OCaml module. This module can then be passed to an OCaml functor, `Corecursive`, that builds the desired function. We discuss the structure of `Corecursive` later in this section.

The functor  $F$  is represented by a parameterized type `'b f`, and the coalgebra  $C$  and the algebra  $A$  are simply defined by types `coalgebra` and `algebra`, respectively. This allows to specify  $\gamma$  naturally as a function from `coalgebra` to `coalgebra f`, and  $\alpha$  as a function from `algebra f` to `algebra`. In the free variable example, if `VarSet` is a module implementing sets of strings, this is done as:

```
type 'b f = I1 of string | I2 of 'b * 'b
type coalgebra = Var of string
                | App of coalgebra * coalgebra
                | Lam of string * coalgebra
type algebra = VarSet.t

let gamma (c:coalgebra) : coalgebra f =
  match c with
  | Var v -> I1 v
  | App(c1, c2) -> I2(c1, c2);;
let alpha (s:algebra f) : algebra =
  match s with
  | I1 v -> VarSet.singleton v
  | I2(s1, s2) -> VarSet.union s1 s2
```

Variables are represented by strings and fresh variables are generated with a counter. Equations are of the form `variable = t`, where the variables on the left-hand side are elements of the domain and the terms on the right side are built up from the constructors of the datatype, constants and variables.

For instance, in the `fv` example, the domain was specified by the following datatype:

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

Recall the four equations above defining the free variables of the  $\lambda$ -term from the introduction:

```
fv t = (fv x) ∪ (fv u)
fv u = (fv y) ∪ (fv t)
fv x = {x}
fv y = {y}
```

A variable name is generated to the application of `fv` to each element of the coalgebra encountered. For example, here we write `v1` for the unknown corresponding to the value of `fv t`; `v2` for `x`; `v3` for `u`; and `v4` for `y`. An equation is represented as a pair of a variable and an element of type `f variable`. The intuitive, informal meaning of a pair  $(v, w)$  is the equation  $v = \alpha(w)$ . For example, in the example above, the equations would be represented as:

```
("v1", I2("v2", "v3")) representing v1 = v2 ∪ v3
("v2", I1("x"))         representing v2 = {x}
("v3", I2("v4", "v1")) representing v3 = v4 ∪ v1
("v4", I1("y"))         representing v4 = {y}
```

The function `solve` can now be described: its arguments are a variable  $v$  for which we want a solution, and a system of equations in which  $v$  appears. It returns a value for  $v$  that satisfies the equations. In most cases the solution is not unique, and specifying `solve` allows the programmer to specify which solution to pick.

For technical reasons, two more functions need to be provided. The function `equal` provides an equality on the coalgebra, which allows the equation generator to know when it has encountered a loop. In most cases, this equality is just the OCaml physical equality `==`; this is necessary because the OCaml equality `=` on coinductive elements does not terminate. In some other cases the function `equal` is an equality function built from both `=` and `==`.

The function `fh` can either be seen as an iterator on the functor `f`, in the style of folding and mapping on lists; or it can be seen as a monadic operator on the functor `f`. It allows the lifting of a function from `'c` (typically `coalgebra`) to `'a` (typically `algebra`) to a function from `'c f` to `'a f`, while folding on an element of type `'e`. It works by destructuring the element of type `'c f` to get zero, one or several elements of type `'c`, successively applying the function on each of them, while passing through the element of type `'e`, and reconstructing an element of type `'a f` with the same constructor used in `'c f`, returned with the final value of the element of type `'e`. For example, in the example on free variables, the function `fh` is defined as:

```
let fh (h: 'c * 'e -> 'a * 'e)
      : 'c f * 'e -> 'a f * 'e = function
| I1 v, e -> I1 v, e
| I2(c1, c2), e -> let a1, e1 = h (c1, e) in
                    let a2, e2 = h (c2, e1) in
                    I2(a1, a2), e2
```

If we had access to an abstract representation of the functor `f`, analyzing it allows to automatically generate the function `fh`. This is what we do in §4.5.

All this is summarized in the signature of a type `SOLVER`, used to specify one of those functions:

```
module type SOLVER =
sig
  type 'b f
  type coalgebra
  type algebra

  val gamma : coalgebra -> coalgebra f
  val alpha : algebra f -> algebra

  type variable = string
  type equation = variable * (variable f)

  val solve : variable -> equation list -> algebra

  val equal : coalgebra -> coalgebra -> bool
  val fh : ('c * 'e -> 'a * 'e)
          -> 'c f * 'e -> 'a f * 'e
end
```

Let us now define the OCaml functor `Corecursive`. From a specification of a function as a module `S` of type `SOLVER`, it generates the equations to be solved and sends them to `S.solve`. Here is how it generates the equations: starting from an element `c` of the coalgebra, it gathers all the elements of the coalgebra that are reachable from `c`, recursively descending with `gamma` and `fh`, and stopping when reaching an element that is equal—in the sense of the function `equal`—to an element that has already been seen. For each of those elements, he generates an associated fresh variable and an associated equation based on applying `gamma` to that element.

From an element `c`, generating the equations and solving them with `solve` returns an element `a` in the coalgebra, the result of applying the function we defined to `c`.

```
module Corecursive :
functor (S: SOLVER) ->
sig
  val main : S.coalgebra -> S.algebra
end
```

We will now explain the default solvers we have implemented and which are available for the programmer to use. These solvers cover the examples we have shown before: a least fixpoint solver, a solver that generates coinductive elements and is used for substitution, and a Gaussian elimination solver.

## 4.2 Least Fixpoint

If the algebra  $A$  is a CPO, then every continuous function  $f$  on  $A$  has a least fixpoint, by the Knaster–Tarski theorem. Moreover, if the CPO satisfies the ascending chain condition (ACC), then this least fixpoint can be computed in finite time by iteration, starting from  $\perp_A$ .

We can apply this technique on the free variables example. The algebra  $A$  equipped with the subset ordering  $(2^{\text{Var}}, \subseteq)$  is a CPO, and its bottom element is  $\perp_A = \{\}$ . It satisfies the ACC as long as we restrict ourselves to the total set of variables appearing in the term. This set is finite because the term is regular and thus has a finite representation.

To implement this, first consider the set of equations: each variable is defined by one equation relating it to the other variables. We keep a guess for each variable, initially set at  $\perp_A$ , and compute a next guess based on the equation for each variable. This eventually converges and we can return the value of the desired variable. Note that to implement this, the programmer needs to check that  $A$  is a CPO with the ascending chain condition, and needs to provide two things:

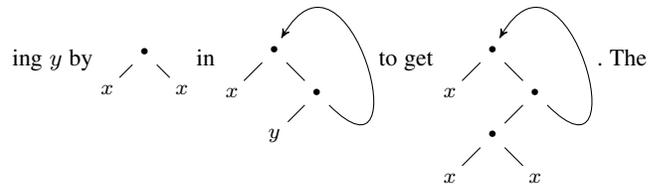
- a bottom element  $\perp_A$ ; and
- an equality on  $A$  that determines when a fixpoint is reached.

The same technique can be used to implement the solver for the abstract interpretation example, as it is also a least fixpoint in a CPO. This CPO is the subset of the join semilattice of abstract domains containing only the elements greater or equal than the input abstract domain. The ACC is ensured by the fact that the abstract domain is always of finite height. The bottom element is the input abstract domain. Much of the code is shared with free variables, and, as pointed out before, only the bottom element of  $A$  and the equality on  $A$  change.

More surprisingly, this technique can also be used in probability examples, where the system of equations looks more like a linear system of equation on  $\mathbb{R}$ . As long as the system of equations is contracting, which is the case in the probability examples (except in some trivial cases), we can solve it by iterative approximation until getting close enough to a fixpoint. The initial element  $\perp_A$  is then 0. The equality on  $A$  is the interesting part: since it determines when to stop iterating, two elements of  $A$  are considered equal if and only if they differ by less than  $\varepsilon$ , where  $\varepsilon$  is the precision of the approximation. Of course, such a linear system could also be solved with Gaussian elimination, as presented below in §4.4.

## 4.3 Generating Coinductive Elements and Substitution

Let us return to the substitution example and the example of replac-



equations we get tell us to find a `v1` such that

```
v1 = App(v2, v3)
v2 = Var("x")
v3 = App(v4, v1)
v4 = App(Var "x", Var "x")
```

Finding such a `v1` is easily done by executing the following code in OCaml:

```
let rec v1 = App(v2, v3)
      and v2 = Var("x")
      and v3 = App(v4, v1)
      and v4 = App(Var "x", Var "x")
in v1
```

that can be easily generated (as a string of text) from the equations. Unfortunately, there is no direct way of generating the element that this code would produce. One workaround is to use the module `Toploop` of OCaml that provides the ability to dynamically execute code from a string, like `eval` in Javascript. But that is not a satisfying solution.

Another solution is to allow the program to manipulate terms by making all subterms mutable using references:

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

This type allows the creation of the desired term by going down the equations and building the terms progressively, backpatching if necessary when encountering a loop. But this is also unsatisfactory, as we had to change the type of `term` to allow references.

The missing piece is mutable variables, which are not present in OCaml. References can simulate them, but they force us to change the type, which we do not want to do. The only way to use and manipulate corecursive types correctly is to have mutable variables. The language constructs we propose should thus be created in a programming language with mutable variables.

The same problem arises whenever  $A$  contains corecursive elements as well, such as in the example on descending sequences.

#### 4.4 Gaussian Elimination

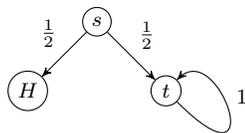
In many of the examples on probabilities and streams, a set of linear equations is generated. For example, the example on probabilistic protocols of §2.3 requires us to find a float `var1` such that

```
var1 = 0.5 + 0.5 * var2
var2 = 0.5 * var1
```

In the case where the equations are contractive, we have already seen that the solution is unique and we can approximate it by iteration. We have also implemented a Gaussian elimination algorithm that can be used to get a more precise answer or when the map is not contractive but the solution is still unique.

But what happens when the linear system has no solution, or an infinite number of solutions? If the system does not have a solution, then there is no fixpoint for the function, and the function is just undefined on that input. If there are an infinite number of solutions, it depends on the application. For example, in the case of computing the probability of heads in a probabilistic protocol, we want the least such solution such that all variables take values between 0 and 1.

For example, let us consider the following probabilistic protocol: Flip a fair coin. If it comes up heads, output heads, otherwise flip again. Ignore the result and come back to this last state, effectively flipping again forever. This protocol can be represented by the following probabilistic automaton:



The probability of obtaining heads, starting from  $s$  and  $t$ , respectively, is given by:

$$\Pr_H(s) = \frac{1}{2} + \frac{1}{2} \cdot \Pr_H(t) \quad \Pr_H(t) = 1 \cdot \Pr_H(t).$$

The set of solutions for these equations for  $\Pr_H(t)$  is the interval  $[0, 1]$ , thus the set of solutions for  $\Pr_H(s)$  is the interval  $[\frac{1}{2}, 1]$ . The desired result, however, is the least of those solutions, namely

$\frac{1}{2}$  for  $\Pr_H(s)$ , because the protocol halts with result heads only with probability  $\frac{1}{2}$ .

#### 4.5 Automatic Partitioning

Providing all the elements to a SOLVER module requires from the programmer a good understanding of the concepts explained in this paper, and a method to solve equations. On the other hand, examples show that the same solving techniques arise again and again. Ideally, what we would like the programmer to write for free variables is just:

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

let rec_lfp fv = function
  | Var v -> {v}
  | App (t1,t2) -> (fv t1) ∪ (fv t2)
  | Lam (x,t) -> (fv t) - {x}
```

where the keyword `rec_lfp` has replaced the keyword `rec`, and signifies that the equations generated for this function should be solved using a least fixpoint algorithm by iteration, as described in §4.2.

This definition is almost enough to generate the SOLVER module. Only three more things need to be specified by the programmer:

- the function `equal` on coalgebras, here just `==`, as in most cases; and
- the two elements allowing to run the least fixpoint algorithm: a bottom element  $\perp_A$  and an equality  $=_A$  on the algebra  $A$ , written `algebra` in the code.

The other elements can be directly computed from a careful analysis of the code of `fv`:

- `fv` can be typed with the usual typing rules on recursive functions. Then `algebra` is defined as its input type and `coalgebra` as its output type;
- a dataflow analysis and analysis of the life of variables in `fv` determines what is executed before the recursive calls and what is executed after the recursive calls (assuming all the recursive calls can be parallelized together). What is executed before the recursive call forms the function `gamma`, and what is executed after the recursive call forms the function `alpha`. Analyzing the arguments that are passed to the recursive calls, as well as the variables that are still alive across the boundary between `gamma` and `alpha`, we build the functor `f`;
- folding on the abstract syntax tree of `f`, we generate code for `fh`; more precisely, we can define `fh` by induction on the structure of the abstract syntax tree defining `f`: if the definition of `'a` `f` does not contain any `'a`, it just returns its second argument; if it is a product, we apply its first argument `h` to every element of type `'a` in the product, passing through the element of type `'e`, and returning a reconstructed product of the results; if it is a sum type, we separate the different cases with a `match`, treat each case as a product and inject back in the sum type; this is done on the free variables example in §4.1;
- the type equation is always defined in the same way; and
- the `solve` function is generic for all functions solved as a least fixpoint by iteration, just depending on the bottom element and the equality on the algebra.

## 5. Discussion

Coalgebraic (coinductive) datatypes and algebraic (inductive) datatypes are similar in many ways. Nevertheless, there are some important distinctions. Algebraic types have a long history, 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 languages support coalgebraic types—for example, Standard ML and F# do not—and even those that do may not do so adequately.

The most important distinction is that coalgebraic objects can be cyclic, whereas algebraic objects are always well-founded. Functions defined by structural recursion on well-founded data always terminate and yield a value under the standard semantics of recursion, but not so on coalgebraic data. A more subtle distinction is that 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.

Despite these differences, there are some strong similarities. They are defined in the same way by recursive type equations, algebraic types as initial solutions and coalgebraic types as final solutions. Because of this similarity, we would like to program with them in the same way, using constructors and destructors and writing recursive definitions using pattern matching.

In this paper we have shown through several examples that this approach to computing with coalgebraic types is not only useful but viable. For this to be possible, it is necessary to circumvent the standard semantics of recursion, and we have demonstrated that this obstacle is not insurmountable. We have proposed new programming language features that would allow the specification of alternative solutions and methods to compute them, and we have given mock-up implementations that demonstrate that this approach is feasible.

The chief features of our approach are the automatic extraction of a finite system of equations from the function definition and its (cyclic) argument, a means for specifying an equation solver, and an interface between the two. In many cases, such as an iterative fixpoint on a codomain satisfying the ascending chain condition, the process can be automated, requiring little extra work on the part of the programmer.

We have also included some theoretical results that clarify and slightly generalize some results of [2].

We have mentioned that mutable variables are essential for manipulating coalgebraic data. Current functional languages in the ML family do not support mutable variables; thus true coalgebraic data can only be constructed explicitly using **let rec**, not programmatically. Moreover, once constructed, it cannot be changed. These restrictions currently render coalgebraic datatypes all but useless. One workaround is to simulate mutable variables with references, but this is a supremely ugly alternative that flies in the face of algebraic elegance. A future endeavor is to provide a smoother and more realistic implementation of these ideas in an ML-like language with mutable variables.

## Acknowledgments

We are grateful to Bob Constable, Nate Foster, Helle Hvid Hansen, Bart Jacobs, Andrew Myers, Stefan Milius, and Ross Tate for helpful comments and pointers to the literature. We would also like to thank Edgar Friendly, Jonathan Kimmitt and Xavier Leroy for useful answers on the `caml-list` discussion group. Part of this work was done while the first two authors were visiting Radboud University Nijmegen and the CWI Amsterdam.

## References

- [1] J. Adámek, S. Milius, and J. Velebil. Elgot algebras. *Log. Methods Comput. Sci.*, 2(5:4):1–31, 2006.
- [2] J. Adámek, D. Lücke, and S. Milius. Recursive coalgebras of finitary functors. *Theoretical Informatics and Applications*, 41:447–462, 2007.
- [3] S. L. Bloom and Z. Ésik. *Iteration Theories: The Equational Logic of Iterative Processes*. Springer-Verlag, New York, 1993.
- [4] V. Capretta, T. Uustalu, and V. Vene. Corecursive algebras: A study of general structured corecursion. In M. V. M. Oliveira and J. 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] S. Chong. Lecture notes on abstract interpretation, 2010. URL <http://www.seas.harvard.edu/courses/cs152/2010sp/lectures/lec20.pdf>. Harvard University.
- [6] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages*, pages 238–252, Los Angeles, 1977. ACM Press, New York.
- [7] A. Eppendahl. Coalgebra-to-algebra morphisms. *Electronic Notes in Theoretical Computer Science*, 29, 1999.
- [8] D. Harel and D. Kozen. A programming language for the inductive sets, and applications. *Information and Control*, 63(1/2):118–139, 1984.
- [9] A. Joyal, R. Street, and D. Verity. Traced monoidal categories. *Math. Proc. Cambridge Phil. Soc.*, 119:447–468, 1996.
- [10] D. Kozen. Realization of coinductive types. *Electr. Notes Theor. Comput. Sci.*, 276:237–246, 2011.
- [11] J. Lambek. A fixpoint theorem for complete categories. *Mathematische Zeitschrift*, 103:151–161, 1968.
- [12] S. Mac Lane. *Categories for the Working Mathematician*. Springer, 1971.
- [13] P. Taylor. *Practical Foundations of Mathematics*. Number 59 in Cambridge Studies in Advanced Mathematics. Cambridge University Press, 1999.
- [14] G. Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, Cambridge, Massachusetts, USA, 1993.

### A. Proof of Theorem 3.1

In this section we present a proof of Theorem 3.1 which extends a result of [2]. In our proof, we make use of an explicit construction of final coalgebras introduced by the second author in [10]. To make this paper self-contained we recall the main definitions and results from [10].

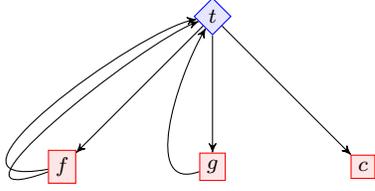
#### A.1 Directed Multigraphs

A *directed multigraph* is a structure  $G = (V, E, \text{src}, \text{tgt})$  with nodes  $V$ , edges  $E$ , and two maps  $\text{src}, \text{tgt} : E \rightarrow V$  giving the source and target of each edge, respectively. We write  $e : s \rightarrow t$  if  $s = \text{src } e$  and  $t = \text{tgt } e$ . When specifying multigraphs, we will sometimes use the notation  $s \xrightarrow{n} t$  for the metastatement, “There are exactly  $n$  edges from  $s$  to  $t$ .”

A *path* is a finite alternating sequence of nodes and edges

$$s_0 e_0 s_1 e_1 s_2 \cdots s_{n-1} e_{n-1} s_n,$$

$n \geq 0$ , such that  $e_i : s_i \rightarrow s_{i+1}$ ,  $0 \leq i \leq n - 1$ . These are the arrows of the free category generated by  $G$ . The *length* of a path is the number of edges. A path of length 0 is just a single node. The first and last nodes of a path  $p$  are denoted  $\text{src } p$  and  $\text{tgt } p$ , respectively. As with edges, we write  $p : s \rightarrow t$  if  $s = \text{src } p$  and  $t = \text{tgt } p$ .



**Figure 1.** A multigraph representing a single-sorted algebraic signature. Blue diamonds represent existential nodes and red squares universal nodes.

A *multigraph homomorphism*  $\ell : G_1 \rightarrow G_2$  is a map  $\ell : V_1 \rightarrow V_2, \ell : E_1 \rightarrow E_2$  such that if  $e : s \rightarrow t$  then  $\ell(e) : \ell(s) \rightarrow \ell(t)$ . This lifts to a functor on the free categories generated by  $G_1$  and  $G_2$ .

## A.2 Type Signatures

A *type signature* is a directed multigraph  $F$  along with a designation of each node of  $F$  as either *existential* or *universal*. The existential and universal nodes correspond respectively to coproduct and product constructors. The directed edges of the graph represent the corresponding destructors.

For example, consider an algebraic signature consisting of a binary function symbol  $f$ , a unary function symbol  $g$ , and a constant  $c$ . This would ordinarily be represented by the polynomial endofunctor  $F = -^2 + - + \mathbb{1}$ , or in OCaml by

```
type t = F of t * t | G of t | C
```

We would represent this signature by a directed multigraph consisting of four nodes  $\{t, f, g, c\}$ , of which  $t$  is existential and  $f, g, c$  are universal, along with edges

$$t \xrightarrow{1} f \quad t \xrightarrow{1} g \quad t \xrightarrow{1} c \quad f \xrightarrow{2} t \quad g \xrightarrow{1} t.$$

The multigraph is illustrated in Fig. 1.

## A.3 Coalgebras and Realizations

Let  $F$  be a type signature with nodes  $V_F$ . An *F-coalgebra* is a  $V_F$ -indexed collection of pairs  $(C_s, \gamma_s)$ , where the  $C_s$  are sets and the  $\gamma_s$  are set functions

$$\gamma_s : C_s \rightarrow \begin{cases} \sum_{\text{src } e=s} C_{\text{tgt } e}, & \text{if } s \text{ is existential,} \\ \prod_{\text{src } e=s} C_{\text{tgt } e}, & \text{if } s \text{ is universal.} \end{cases}$$

A morphism of  $F$ -coalgebras is a  $V_F$ -indexed collection of set maps  $h_s$  that commute with the  $\gamma_s$  in the usual way. This corresponds to the traditional definition of an  $F$ -coalgebra for an endofunctor  $F$  on  $\text{Set}^V$ .

Coalgebras are equivalent to *realizations*. An *F-realization* is a directed multigraph  $G$  along with a multigraph homomorphism  $\ell : G \rightarrow F$ , called a *typing*, with the following properties.

- If  $\ell(u)$  is existential, then there is exactly one edge of  $G$  with source  $u$ .
- If  $\ell(u)$  is universal, then  $\ell$  is a bijection between the edges of  $G$  with source  $u$  and the edges of  $F$  with source  $\ell(u)$ .

A homomorphism of  $F$ -realizations is a multigraph homomorphism that commutes with the typings.

**THEOREM A.1** ([10]). *The categories of F-coalgebras and F-realizations are equivalent (in the sense of [12, §IV.4]).*

## A.4 Final Coalgebras

Realizations allow us to give a concrete construction of final coalgebras that is reminiscent of the Brzozowski derivative on sets

of strings. Here, instead of strings, the derivative acts on certain sets of paths of the type signature.

Let  $F$  be a type signature. Construct a realization  $R_F, \ell_F$  as follows. A node of  $R_F$  is a set  $A$  of finite paths in  $F$  such that

- $A$  is nonempty and prefix-closed;
- all paths in  $A$  have the same first node, which we define to be  $\ell_F(A)$ ;
- if  $p$  is a path in  $A$  of length  $n$  and  $\text{tgt } p$  is existential, then there is exactly one path of length  $n + 1$  in  $A$  extending  $p$ ;
- if  $p$  is a path in  $A$  of length  $n$  and  $\text{tgt } p$  is universal, then all paths of length  $n + 1$  extending  $p$  are in  $A$ .

The edges of  $R_F$  are defined as follows. Let  $A$  be a set of paths in  $F$  and  $e$  an edge of  $F$ . Define the *Brzozowski derivative* of  $A$  with respect to  $e$  to be

$$D_e(A) = \{p \mid (\text{src } e) e p \in A\},$$

the set of paths obtained by removing the initial edge  $e$  from paths in  $A$  that start with that edge. If  $A$  is a node of  $R_F$  and  $D_e(A)$  is nonempty, we include exactly one edge

$$\langle A, e \rangle : A \rightarrow D_e(A)$$

in  $R_F$  and take  $\ell_F(\langle A, e \rangle) = e$ . It is readily verified that  $\text{tgt } \langle A, e \rangle = D_e(A)$  satisfies properties (i)–(iv) and that  $\ell_F(D_e(A)) = \text{tgt } e$ , so  $\ell_F$  is a typing.

**THEOREM A.2.** *The realization  $R_F, \ell_F$  is final in the category of F-realizations. The corresponding F-coalgebra as constructed in Theorem A.1 is final in the category of F-coalgebras.*

## A.5 F-Algebras

An *F-algebra* is a  $V_F$ -indexed collection of pairs  $(A_s, \alpha_s)$ , where the  $A_s$  are sets and the  $\alpha_s$  are set functions

$$\alpha_s : \begin{cases} \left( \sum_{\text{src } e=s} A_{\text{tgt } e} \right) \rightarrow A_s, & \text{if } s \text{ is existential,} \\ \left( \prod_{\text{src } e=s} A_{\text{tgt } e} \right) \rightarrow A_s, & \text{if } s \text{ is universal.} \end{cases}$$

A morphism of  $F$ -algebras is a  $V_F$ -indexed collection of set maps  $h_s$  that commute with the  $\alpha_s$  in the usual way. This corresponds to the traditional definition of an  $F$ -algebra for an endofunctor  $F$  on  $\text{Set}^V$ .

## A.6 Proof of the Theorem 3.1

An  $F$ -coalgebra-algebra morphism is a set function  $h : C \rightarrow A$ , where  $(C, \gamma)$  is an  $F$ -coalgebra and  $(A, \alpha)$  is an  $F$ -algebra, such that the following diagram commutes:

$$\begin{array}{ccc} C & \xrightarrow{h} & A \\ \gamma \downarrow & & \uparrow \alpha \\ FC & \xrightarrow{Fh} & FA \end{array} \quad (7)$$

**LEMMA A.3** (Lambek's Lemma [11]). *The structure map of the initial F-algebra is invertible. The structure map of the final F-coalgebra is invertible.*

An  $F$ -realization  $G = (V, E, \text{src}, \text{tgt}, \ell)$  is *well-founded* if all directed  $E$ -paths are finite. An  $F$ -coalgebra is *well-founded* if its corresponding  $F$ -realization is.

**LEMMA A.4.** *Every F-coalgebra contains a maximal well-founded subcoalgebra.*

*Proof.* Equivalently, every  $F$ -realization  $G = (V, E, \text{src}, \text{tgt}, \ell)$  contains a maximal well-founded  $F$ -subrealization  $\text{wf } G$ . The nodes  $\text{wf } V$  are the nodes of  $G$  from which there are no infinite directed  $E$ -paths. The graph  $\text{wf } G$  is the induced subgraph on  $\text{wf } V$ . Equivalently, the set of nodes of  $\text{wf } G$  is the smallest set of nodes  $A$  of  $G$  satisfying the closure condition: if all  $E$ -successors of  $s$  are in  $A$ , then  $s \in A$ .  $\square$

The induction principle for well-founded  $G = (V, E, \text{src}, \text{tgt}, \ell)$  is:

$$\frac{\forall x (\forall y \in \mathbf{n}(x) P(y)) \rightarrow P(x)}{\forall x P(x)}, \quad (8)$$

where  $\mathbf{n}$  is the successor function on  $G$ ; that is,  $\mathbf{n}(x) = \{\text{tgt } e \mid e \in E, \text{src } e = x\}$ .

LEMMA A.5. *Let  $R_F = (V, E, \text{src}, \text{tgt}, \ell)$  be the final  $F$ -realization. Then  $\text{wf } R_F$  is an  $F$ -algebra.*

**Remark** We will show in Corollary A.7 that  $\text{wf } R_F$  is in fact the initial  $F$ -algebra (up to isomorphism). To show initiality, we need to show that there is a unique  $F$ -algebra morphism to any other  $F$ -algebra. This will follow as a special case of Theorem 3.1(iv) below.

*Proof.* By Lemma A.3, the structure map  $(\gamma_s \mid s \in V_F)$  of the final  $F$ -coalgebra corresponding to  $R_F$  is invertible, thus forms an  $F$ -algebra. Translating back to the realization  $R_F$ , this means that

- for every edge  $e \in E_F$  such that  $\text{src } e$  is existential and every node  $v$  of  $R_F$  with  $\ell(v) = \text{tgt } e$ , there exists a unique node  $u$  and edge  $d$  of  $R_F$  such that  $\text{src } d = u$ ,  $\text{tgt } d = v$ , and  $\ell(d) = e$ ; and
- for every universal node  $s \in V_F$  and tuple  $(v_e \mid \text{src } e = s)$  of nodes of  $R_F$  such that  $\ell(v_e) = \text{tgt } e$ , there exist a unique node  $u$  and tuple of edges  $(d_e \mid \text{src } e = s)$  of  $R_F$  such that  $\text{src } d_e = u$ ,  $\text{tgt } d_e = v_e$ , and  $\ell(d_e) = e$ .

The existence and uniqueness of  $u$  in the above two cases assert the closure of  $R_F$  under the algebraic operations. The subrealization  $\text{wf } R_F$  is closed under these operations, because any node all of whose immediate  $E$ -successors are in  $\text{wf } R_F$  is also in  $\text{wf } R_F$ , therefore  $\text{wf } R_F$  is a subalgebra of  $R_F$ .  $\square$

We are now ready to present the proof of Theorem 3.1, which we recall next, adding an extra item on parametric recursive coalgebras to match the theorem of Adamek, Luecke and Milius.

THEOREM A.6. *Let  $(C, \gamma)$  be an  $F$ -coalgebra. The following are equivalent:*

- (i)  $C$  is well-founded.
- (ii) The induction principle (8) is valid for  $C$ .
- (iii) There is a unique coalgebra morphism  $C \rightarrow \text{wf } R_F$ .
- (iv) There is a unique coalgebra-algebra morphism from  $C$  to any  $F$ -algebra.
- (v) There is a unique parameterized coalgebra-algebra morphism from  $C$  to any  $F$ -algebra.

*Proof.* The equivalence of (i) and (ii) is a fundamental property of relational algebra. The implication (i)  $\Rightarrow$  (ii) requires the axiom of dependent choice.

Assuming (i) and (ii), (iv) can be proved by defining a coalgebra-algebra morphism by induction, using (8). Let  $(A_s, \alpha_s)$  be an arbitrary  $F$ -algebra. Assume the coalgebra  $C$  is given in the form of an  $F$ -realization  $G = (V, E, \text{src}, \text{tgt}, \ell)$ . We must define maps  $h_s : \ell^{-1}(s) \rightarrow A_s$  for  $s \in V_F$  satisfying condition (7). This is equivalent to the following two conditions. Let  $s \in V_F$  and  $u \in V$  such that  $\ell(u) = s$ .

- If  $s$  is existential, let  $d$  be the unique edge with  $\text{src } d = u$ , let  $v = \text{tgt } d$ , and let  $e = \ell(d)$ . Then

$$h_s(u) = \alpha_s(\text{in}_e(h_{\text{tgt } e}(v))) \in A_s.$$

- If  $s$  is universal, for each  $e$  such that  $\text{src } e = s$ , let  $d_e$  be the unique edge with  $u = \text{src } e$  and  $\ell(d_e) = e$ , and let  $v_e = \text{tgt } d_e$ . Then

$$h_s(u) = \alpha_s(h_{\text{tgt } e}(v_e) \mid \text{src } e = s) \in A_s.$$

The maps  $h_s$  are uniquely defined by these equations due to the well-foundedness of the  $E$ -successor relation on  $G$ .

By Lemma A.5,  $\text{wf } R_F$  is an  $F$ -algebra, thus (iii) follows as a special case of (iv).

To argue that (iii) implies (i), we observe that under any morphism of  $F$ -realizations  $C \rightarrow \text{wf } R_F$ , an infinite path in  $C$  would map to an infinite path in  $\text{wf } R_F$ , which cannot exist by definition, since  $\text{wf } R_F$  is well-founded. Thus  $C$  must be well-founded as well.

We will now show (v)  $\Leftrightarrow$  (iv).

We first show (v)  $\Rightarrow$  (iv). Suppose that there is a unique parameterized coalgebra-algebra morphism from  $C$  to any  $F$ -algebra. That is, for any  $\alpha' : FA \times C \rightarrow A$  there is a unique  $h$  which makes the following diagram commute:

$$\begin{array}{ccc} C & \xrightarrow{\quad h \quad} & A \\ \langle \gamma, \text{id} \rangle \downarrow & & \uparrow \alpha' \\ FC \times C & \xrightarrow{\quad Fh \times \text{id} \quad} & FA \times C \end{array} \quad (9)$$

We want to show that there is a unique coalgebra-algebra morphism from  $C$  to any  $F$ -algebra.

Take an arbitrary  $F$ -algebra  $\alpha : A \rightarrow FA$  and consider  $\alpha' = \alpha \circ \pi_1 : FA \times C \rightarrow A$ . Using diagram (9), we know that there exists a unique  $h : C \rightarrow A$  such that  $h = \alpha \circ \pi_1 \circ (Fh \times \text{id}) \circ \langle \gamma, \text{id} \rangle$ . We show that  $h$  is a coalgebra-algebra morphism from  $C$  to  $A$  and, moreover, that it is unique.

$$\begin{aligned} h &= \alpha \circ \pi_1 \circ (Fh \times \text{id}) \circ \langle \gamma, \text{id} \rangle && \text{diagram (9)} \\ &= \alpha \circ Fh \circ \pi_1 \circ \langle \gamma, \text{id} \rangle && \pi_1 \text{ is a natural transf.} \\ &= \alpha \circ Fh \circ \gamma && \pi_1 \circ \langle f, g \rangle = f \end{aligned}$$

For uniqueness note that any other coalgebra-algebra morphism  $g$  from  $C$  to  $A$  also makes diagram (9) commute, for  $\alpha' = \alpha \circ \pi_1$ :

$$\begin{aligned} g &= \alpha \circ Fg \circ \gamma && \text{definition of coalgebra-algebra} \\ & && \text{morphism} \\ &= \alpha \circ Fg \circ \pi_1 \circ \langle \gamma, \text{id} \rangle && \pi_1 \circ \langle f, g \rangle = f \\ &= \alpha \circ \pi_1 \circ (Fg \times \text{id}) \circ \gamma && \pi_1 \text{ is a natural transformation} \end{aligned}$$

Hence,  $g = h$ .

For the converse implication, we need the following fact. Let  $\gamma : C \rightarrow FC$  be an  $F$ -coalgebra. Define  $G(X) = C \times FX$ . Now note that if  $(C, \gamma)$  is a well-founded  $F$ -coalgebra then  $(C, \langle \gamma, \text{id} \rangle)$  is a well-founded  $G$ -coalgebra. Hence, we have that (i) for  $F$  implies (i) for  $G$  which in turn (because we know, from what we proved above, that (i) and (iv) are equivalent for a given functor) implies (iv) for  $G$ . Now, assuming we have (iv) for  $G$ , (v) follows trivially for  $F$  (note that diagram (9) for  $F$  is a coalgebra-algebra morphism diagram for  $G$ ).  $\square$

COROLLARY A.7. *The  $F$ -coalgebra  $\text{wf } R_F$  is (up to isomorphism) the initial  $F$ -algebra.*

*Proof.* The structure  $\text{wf } R_F$  is an  $F$ -algebra by Lemma A.5. But it is also a well-founded  $F$ -coalgebra by definition. By the equivalence of Theorem 3.1(i) and (iv), there is a unique  $F$ -algebra morphism from  $\text{wf } R_F$  to any  $F$ -algebra, thus  $\text{wf } R_F$  is initial.  $\square$